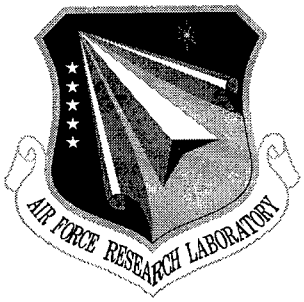
AFRL-IF-RS-TR-1999-236
Final Technical Report
October 1999



SURVIVABILITY IN OBJECT SERVICE ARCHITECTURES (OSA)

Object Services and Consulting, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E292

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-236 has been reviewed and is approved for publication.

APPROVED: Patrick M. Hurley

PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Jr., Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGA, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

SURVIVABILITY IN OBJECT SERVICE ARCHITECTURES

David Wells
Steve Ford
David Langworthy
Thomas Bannon
Nancy Wells
Venu Vasudevan

Contractor: Object Services and Consulting, Inc.
Contract Number: F30602-96-C-0330
Effective Date of Contract: 01 July 1996
Contract Expiration Date: 31 May 1999
Short Title of Work: Survivability in Object Service Architectures

Period of Work Covered: Jul 96 – May 99

Principal Investigator: David Wells
Phone: (410) 318-8938
AFRL Project Engineer: Patrick Hurley
Phone: (315) 330-3624

Approved for public release; distribution unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Patrick Hurley, AFRL/IFGA, 525 Brooks Rd, Rome, NY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Oct 99	Final Jul 96 - May 99

4. TITLE AND SUBTITLE

SURVIVABILITY IN OBJECT SERVICE ARCHITECTURES (OSA)

5. FUNDING NUMBERS

C - F30602-96-C-0330
PE - 62301E
PR - E017
TA - 01
WU - 07

6. AUTHOR(S)

David Wells, Steve Ford, David Langworthy, Thomas Bannon, Nancy Wells and Venu Vasudevan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Object Services and Consulting, Inc.
6111 Baywood Ave.
Baltimore, MD 21209-3803

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFGA
525 Brooks Rd
Rome, NY 13441-4505

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-236

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Patrick M. Hurley, IFGA, 315-330-3624

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The military of the future will increasingly rely upon "information superiority" to dominate the battlespace. The size and complexity of the software systems necessary to achieve this goal makes them highly vulnerable to the loss or degradation of hosts, networks, or processed due to physical and information warfare attacks, hardware and infrastructure failures, and software errors. This report summaries the goals and results of a project that developed an architecture and software mechanisms to make military and commercial software applications based on the popular Object Services Architecture (e.g., OMG's CORBA) model far more survivable than is currently possible, while at the same time maintaining the flexibility and ease of construction that characterizes OSA-based applications.

14. SUBJECT TERMS

Common Object Request Broker Architecture (CORBA), Distributed Information Systems (DIS), Object Services Architecture (OSA)

15. NUMBER OF PAGES

176

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Executive Summary

The military of the future will increasingly rely upon "information superiority" to dominate the battlespace. The size and complexity of the software systems necessary to achieve this goal makes them highly vulnerable to the loss or degradation of hosts, networks, or processes due to physical and information warfare attacks, hardware and infrastructure failures, and software errors.

This report summarizes the goals and results of a project that developed an architecture and software mechanisms to make military and commercial software applications based on the popular Object Services Architecture (e.g., OMG's CORBA) model far more survivable than is currently possible, while at the same time maintaining the flexibility and ease of construction that characterizes OSA-based applications.

Problem Statement

The military of the future will increasingly rely upon "information superiority" to dominate the battlespace. Achieving information superiority will require software applications that are far larger, far more complex, and far more distributed than comparable applications in existence today. The size and complexity of such systems makes them highly vulnerable to the loss or degradation of hosts, networks, or processes due to physical and information warfare attacks, hardware and infrastructure failures, and software errors. Since loss or degradation is inevitable, it is essential that such systems behave well when this occurs. A system that can repair itself or degrade gracefully to preserve as much critical functionality as possible in the face of attacks and failures is called a survivable system.

Objective

This project has developed software mechanisms to ensure the survivability of such systems that go well beyond the traditional approaches of fault tolerance and replicated services. Those techniques, while valuable, are in themselves insufficient to respond to the full range of problems that can face a system since they create "islands of availability" but do nothing to address system-wide concerns. The following two examples illustrate the kinds of issues addressed by our survivability work.

Mission planning for a sortie in regional conflict with multiple coalition partners requires many resources, among them a map server. Assume the local map server becomes unavailable and that the backup map server is located at a remote location and reachable only over slow communication lines. There is a coalition map server available with good performance characteristics, but its data is considered to be of lower quality and the labels are specified in a foreign language. Under many circumstances, it would be desirable to use the coalition map server, but existing systems cannot switch an active connection and are limited to exact substitutes for a service. A survivable system needs to be able to switch compatible services in an established connection and substitute acceptable alternatives.

The ability to substitute services is only one aspect of survivability. Consider an information warfare attack focused on NT machines. As the NT machines begin to fail, essential processing must be moved over to UNIX machines. This in turn requires terminating or delaying non essential processing on those machines. However, there are many different threats, each with its own optimal response, and more than one threat may materialize at the same time. Addressing this in an ad hoc manner is not possible. A survivable system must be able to dynamically adapt to the threats in its environment to reallocate essential processing to the most robust resources.

Approach

This project is developing software mechanisms to make military and commercial software applications based on the popular Object Services Architecture (e.g., OMG's CORBA) model far more survivable than is currently possible, while at the same time maintaining the flexibility and ease of construction that characterizes OSA-based applications.

The keys to making systems survivable are:

- design patterns with well designed "joints" between components to make reconfiguration possible
- monitors to detect when failure or degradation has occurred
- models of client needs and resource capabilities to allow alternate configurations to be found
- multiple reconfiguration strategies with different "goodness" characteristics applicable in different situations
- the ability to select and execute a good reconfiguration strategy that attempts to satisfy application requirements, make good
- use of resources, and avoid vulnerable configurations

We have developed a comprehensive approach to satisfying these requirements consisting of: a survivable object abstraction in which survivable services and applications can be developed; a collection of models describing capabilities, needs, and threats; and the architecture of a Survivability Service that manages the survival of systems constructed in accordance with the survivable object abstraction. Our goal was to demonstrate the feasibility of this approach by building and demonstrating a prototype Survivability Service consisting of all of the above capabilities except monitoring, which we assume will be developed elsewhere. To maximize the utility of the Survivability Service, we leveraged related work such as fault tolerance techniques, OMG CORBA & Object Services, failure detectors, and various system models. We have proposed the Survivability Service specification to the Object Management Group and plan to make the prototype available as a reference implementation.

Technical Results

All reports listed below are included in the Appendix to this document and are publicly available on our project Website at:

http://www.objs.com/Survivability/index.html

Documents

Survivability in OSA Architectures - 1997

This report describes the goals, approach, and anticipated results of the project "Survivability in Object Services Architectures". It also introduces a collection of other reports produced on the project.

Composition Model for Object Services Architectures - 1997

This report describes the static properties of a survivable object abstraction that extends standard object models in several ways to allow objects and applications to be reconfigured to recover from or protect against failures. Survivable objects are the basis for constructing survivable applications.

Evolution Model for Object Services Architectures - 1997

This report describes the dynamic properties of the survivable object abstraction (introduced in OSA Composition Model) that it possible to safely migrate a running application from one legitimate configuration into another legitimate configuration. Both semantically identical and semantically similar transformations are possible under this model, which allows applications to continue to survive in degraded mode when system resources become unavailable due to attack or failure.

OSA Survivability Service - 1998

This report describes the architecture of a Survivability Service that manages survivable objects and applications constructed using the survivable object abstraction. The Survivability Service is compatible with existing work in failure detection and classification, fault tolerance, and highly available systems. Portions of the Survivability Service are being prototyped as part of this project.

QoS & Survivability - 1998

This report describes recent research in service-level quality of service and the relationship between survivability and quality of service.

Notes on a Command Post Scenario - 1998

This report is a working paper describing the likely software environment of a future military command post, the connectivity between a command post and its outside environment, and a typical activity that takes place within the command post. This will be used to derive the survivability requirements of a command post and define a scenario for demonstrating the Survivability Service.

Survivability is Utility - 1998

This report explores how Utility Theory (a sub-discipline of microeconomics) can be exploited to define metrics to evaluate the successfulness of survivable systems and can be used by Survivability Management Systems to plan actions to ensure system survivability.

Estimating Service Failure - 1999

In the process of creating and maintaining survivable configurations, the Survivability Service needs to predict the likelihood that, within some time interval, a service will be damaged to an extent that it cannot provide the required level of service. This paper

discusses a basic model of how services are provided by resources, how threats against those services are modeled, and how the probability of service failure is computed from the threat model.

Software - OSA Survivability Toolkit

The OSA Survivability Toolkit is a collection of mechanisms that implement some of the capabilities defined for the Survivability Service. In the current release, the following components have been implemented.

OBJS Market v2.0 - April 1999

The Survivability Service takes a market-based approach to resource allocation when constructing survivable configurations and when reconfiguring in response to negative events such as resource failure or degradation, positive events such as resources coming back on line, or neutral events such as a change in workload or the relative priority of activities.

OBJS Rebinding Mechanism v1.0 - April 1999

The OBJS Object Service Rebinding Mechanism v1.0 is a sample implementation of a mechanism to switch the binding of a CORBA (Orbix) client from one server to another, without any action on the part of the client or server, and while the client and servers remain running. This mechanism was developed to facilitate the real-time rebinding of clients and servers in CORBA-based software systems in which it was detected or suspected that switching servers on the fly might improve the overall survivability of the system. For instance, the machine on which a specific server is running might be under attack, or the specific implementation of the server might have become vulnerable to attack.

Survivability Desk v1.0 - April 1999

The Survivability Desk is the User Interface through which a system administrator interacts with a Survivability Service. The Survivability Desk provides a view of system status and is the primary way in which the models used by the Survivability Service are populated and modified.

OBJS Ensemble v0.50 - May 1999

Ensemble is Cornell's distributed communication system. Its Maestro Interoperability Tools provide support for Service Replication in the OBJS Survivability Toolkit by enabling CORBA (Orbix) clients to access Maestro group objects via IIOP. This, when combined with the OBJS Rebinding Mechanism, can provide the means the OBJS Market needs to control service replication, rebinding, and migration in the interest of improving system survivability. This port of the Maestro Interoperability Tools to Windows NT was done in the interest of supporting replicated services across platform types.

Survivability Demonstrations

The survivability mechanisms are demonstrated in three independent demonstrations:

- Replica Allocation via Market Mechanisms and Threat Models in the Presence of Failures. A videotape, *Survivability Demonstration,* illustrates the use of a market and a threat model to allocate service replicas in a simulation of an environment consisting of a small, homogenous collection of hosts. Replicas are positioned to minimize the risk of simultaneous failure of all replicas implementing a service. When individual replicas fail, the system automatically rebalances. The need for the market to consider correlated failures is illustrated by an example in which a brittle allocation is made when correlation is ignored. Also illustrated is the ability of the market to dynamically reallocate resources when the relative priority of the services changes as a result of changed user requirements.
- Command Post Scenario. Market-based resource allocation is illustrated by a demonstration included in the software release *OBJS Survivability Desk v1.0.* The demonstration allocates a variety of services in a semi-realistic model of a fictitious command post. A description of the expected structure and operational requirements of such a command post are described in *Notes on a Command Post Scenario.* This demonstration exercises the Survivability Service market and modeling capabilities in a realistic setting, while at the same time eliminating extraneous factors irrelevant to project goals. The demonstration is canned, in the sense that user feedback to affect the course of the demonstration is not possible, although the viewer may browse the underlying system and threat models to obtain a detailed view of market activity.
- Dynamic Service Rebinding. The ability to dynamically rebind running clients and services without the cooperation or knowledge of either client or server is illustrated by a demonstration included with the *OBJS Rebinding Mechanism v1.0.* Service rebinding is an essential component of any survivability strategy, since it enables the reconfiguration of services and applications from an entity, a Survivability Service, outside of the monitored application.

Technology Transition Accomplishments

Object Management Group

Reports documenting the *survivable object abstraction* (*OSA Composition Model* and *OSA Evolution Model*) and the design of a Object Survivability Service (*OSA Survivability Service*) were submitted to the Object management Group (OMG) per contract requirements. These documents have OMG tracking numbers:

- internet/99-04-03: OSA Survivability Service
  Formats: ASCII, RTF, PostScript, PDF, Word
- internet/99-04-02: Evolution Model for Object Services Architecture
  Formats: ASCII, RTF, PDF, Word, PostScript
- internet/99-04-01: Composition Model for Object Services Architecture
  Formats: ASCII, RTF, PostScript, PDF, Word

A search of the OMG documents (http://www.omg.org/cgi-bin/doclist.pl) list indicates that no alternate approaches to survivability have been presented to OMG.

Preliminary results of the project were presented to the OMG Security SIG at an OMG meeting in June, 1998.

Papers

A paper based on our survivability metrics has been solicited by the editorial board of the *DoD Software Tech News* for an issue on software testing. This paper will be completed after the term of this contract, but is based on results obtained during the period of the contract.

Project Website

All project reports are publicly accessible at our project Website at:

http://www.objs.com/Survivability/index.html

Presentations

The project results have been presented at the following workshops and meetings. Copies of the overheads used appear on our Website and were distributed at the meetings.

- Formal Methods Workshop (1996) - DARPA Formal Methods Workshop, Lake Placid, NY, July 1996.
- Rome Lab Technical Exchange (1996) - Rome Laboratory C3AB Technical Exchange Meeting, Utica, NY, December 1996.
- Wrapper Workshop (1997) - DARPA Wrapper Workshop, Lake Tahoe, CA, July 1997.
- Rome Lab Technical Exchange (1997) - Rome Laboratory C3AB Technical Exchange Meeting, Utica, NY, December 1997.
- OMG SecSIG (1998) - Object Management Group Security SIG Orlando, FL, June 1998.

CDRL Log

- A001 – Monthly Progress Reports #1 - #30
- A002 – Monthly DD1586 & Quarterly Financial Graphics
- A003 – Annual Report - submitted to government in 1997
- A004 – *Composition Model for Object Services Architecture* - submitted to government in 1997 & to the Object Management Group (OMG tracking # internet/99-040-01) Also Appendix A-2

- A005 – *Evolution Model for Object Services Architecture* - submitted to government in 1997 & to the Object Management Group (OMG tracking # internet/99-040-02) Also Appendix A-3
- A006 – *OSA Survivability Service* - submitted to government in 1997 & to the Object Management Group (OMG tracking # internet/99-040-03) Also Appendix A-4
- A007 – Presentation Material for Annual Review – submitted at Review
- A008 – Presentation Material for Rome Tech Exchange Meetings – submitted at meetings and attached as appendices to following month's Monthly Progress Reports
- A009 – Software User Manuals for OSA Survivability Toolkit: *Installation and User's Manual: OBJS Rebinding Service - v1.0*, *Installation and User's Manual: Survivability Service Market – v2.0*, *Installation and User's Manual: Survivability Desk – v1.0*, and *Installation and User's Manual for OBJS Ensemble v0.50* – submitted to the government 1999
- A010 – COTS Software Manuals – No COTS software is being delivered under this contract.
- A011 – Final Report (this document)

Lessons Learned

Emergent Survivability Appears Feasible

While considerable work remains to prove that our initial hypothesis that achieving emergent survivability as a system property was correct, we have demonstrated that several of the necessary ingredients are possible and have not yet hit any critical roadblocks. The survivable object abstraction and survivability architecture we developed were enthusiastically received when presented at workshops attended by other DARPA funded researchers and were considered to be a viable framework. Our successful use of a market to allocate resources for survivability based on a threat model is encouraging, since markets are already well known to be a good way to allocate resources without introducing choke points or requiring excessive centralized organization or design. We have implemented or adapted several of the mechanisms required by the design and the remaining mechanisms do not appear to pose any conceptual difficulties. Our projection of trends in Trading and Failure/Intrusion Detection were accurate enough that we are convinced that our design is compatible with developments elsewhere. Our analysis of trends in software Replication were also in the right direction, although that field has not matured as fast as we had expected (see below). Thus, we are convinced that we are heading in the right direction and that usable results are within reach.

Early Scenarios and Scenario Sharing

Our coding efforts were most successful after we were able to scope the development through a semi-realistic scenario. The Survivability Service as designed applies a large number of survivability strategies; a far larger number than can reasonably be implemented in a project of this size. A scenario allowed us to focus on only those techniques required for that scenario. The existence of the scenario also was very useful in debugging and tuning the market behavior, since it is far easier to see that the system is behaving "reasonably" when viewed from the macro level of the scenario than from the micro level of the market decisions.

Developing such a scenario took a long time. It would have been very helpful if research teams were to share their scenarios, even if the scenarios shared are not directly applicable to the needs of other teams. The DARPA-funded Intrusion Detection work that is constructing and making available synthetic message traffic for a fictitious Air Force base is a good example of this; while it is not directly applicable to survivability, it at least provides a reasonable mix of services running in a reasonable network. It would be far easier to expand this base to also cover survivability than to construct a scenario from scratch. Another advantage of shared scenarios would be to make it easier to combine technologies, since solutions would be addressing a common model and running in a common environment. It would not solve the problem completely, but would make it easier.

Common scenarios are becoming fairly common in ISO projects, but ITO projects could benefit from them as well. This is especially true when the smaller budgets of ITO projects are considered. With these smaller budgets, developing a scenario consumes a higher proportion of project resources. Also, because ITO projects are typically further from the end user, this development is fundamentally harder due to lack of domain knowledge. We did not adequately budget for scenario development.

Demonstrating Survivability With Intermediate Results is Hard

Survivability is achieved through an amalgam of a large number of techniques rather than a single "silver bullet". Many of these (e.g., failure detection) were outside the scope of this project, while others, although within the technology scope, could not be implemented without a much larger project. Until a critical mass of technology is developed or acquired, a comprehensive and convincing demonstration of survivability is not possible. A truly interesting demonstration requires the construction of a large number of independent, relatively uninteresting pieces of technology. Thus, getting to a complete intermediate state having intrinsic value for others is very difficult. We suspect that this is a common problem with large software engineering R&D projects in which a new field must be both defined and implemented. A challenge, that we feel we did not identify until very late and thus did not adequately meet, is how to design for a large goal (survivability) while at the same time defining inherently useful intermediate waypoints that are demonstrable and useful should work stop at any of those points. Earlier access to a significant scenario would possibly have gone a long way toward defining these intermediate points (see above).

Using Research Quality Software

We attempted to import replication software, based on our assumption from the longevity of the field, the quantity of published papers, and the (comparatively) large number of systems that have been built that we would be better off importing than building. We went down several paths in trying to incorporate replication. None of them was entirely successful, due either to licensing issues (see below) or the immaturity of the software. The main problems with the software we tried to use were: narrow-path development in which a concept was proven sufficiently to meet limited project goals but not for wider use, failure to keep the software up to date with the evolution of its software environment (e.g., failure to upgrade to CORBA 2.0), and lack of source code that could be analyzed and modified as needed. Version compatibility when using prototype software is a big issue because there is little incentive for the developers to keep the software current; it is not a commercial product and there is no research value in maintaining currency. However, this makes promising looking software rapidly useless as a platform for further development and is a prime cause of projects reinventing software that should be able to be reused. It might be a good idea for DARPA (or someone) to put aside money to improve, port, and maintain selected prototype software for community use. While an initial expense, this might reduce overall costs by reducing the temptation to reimplement in order to have control over code.

Licensing Issues

The more mature replication systems (Isis and systems which rely upon it) became unavailable when Isis' eventual owner withdrew it as a product from the market. Its free predecessor had been withdrawn earlier. This forced us to use less mature software, which introduced its own issues (above).

Conclusions

We feel that the project was quite successful, although we did not get as far as we had hoped. We are quite satisfied with our survivable object abstraction, both in its capabilities and the fact that it is compatible with the major trends in object service technology such as CORBA and Active-X and with much of what is going on in the Java and agents worlds as well. We are also satisfied with our architecture for a Survivability Service and feel that it has a future within the OMG if properly championed. Our use of a combination of predictive threat models and market-based resource allocation is unique to our knowledge; projections of future behavior combined with the ability to rapidly reconfigure are essential to any survivability strategy. We had expected to integrate the components of the Survivability Toolkit into a more comprehensive demonstration, but the unexpectedly large effort involved in selecting a replication mechanism and getting it to work preclude this. The components implemented demonstrate the key ideas and serve as a good basis for future development, but are not ready for "prime time".

A number of open issues were identified and three proposals to continue the work were submitted to DARPA: *Survivability in AITS, Sensitivity Analysis of Self-Adaptive*

Systems, and *Survivability Achieved Via Electronic Markets (SAVEM)*. The main issues are:

- The use of resource, service, and threat models to predict the robustness and (future) utility of a configuration appears necessary to any survivability planning. However, such models are necessarily "noisy" due to a number of unavoidable modeling errors, including ambiguity and delays in detecting and reporting failures/attacks, uncertainty as to the frequency and severity of attacks and failures, and the difficulty in modeling the statistical properties of sophisticated attacks that can be launched when desired in a way to cause the greatest harm. Additional work needs to be done to more accurately model and measure the kinds of errors to which the models are subject, and to determine whether good resource allocation decisions can be made in the presence of such errors or whether the noise overwhelms the decision process. This involves theoretical work to model the noise and pragmatic work to measure behavior in a real system.
- The various models must be partitioned in some way to provide information where it is needed without creating a global choke point that would both degrade performance and provide a tempting target for attack. However, in order to make use of remote resources, some knowledge of the properties of these resources is required. The issue is complicated (or perhaps simplified) by the fact that other partitionings exist. These partitionings may represent ownership, communications, or security (among others). Existing partitions must be respected. Particularly interesting are security partitions that define information boundaries. Threat and resource models are an important form of information. It is difficult to imaging a security administrator being willing to allow information about his system's vulnerabilities flow outside of the security domain, even if to do so is necessary in order to construct optimally survivable systems. Architectural and modeling issues remain in this area, which can best be explored in the context of adding survivability to a larger system.
- Survivability and quality of service are in a sense duals, since both are attempting to provide client processes with the services they need. The two fields approach the problem from different directions: QoS is primarily concerned with accurately specifying what it takes to satisfy a client's needs and then providing techniques to meet those goals, while survivability concerns itself with models and techniques to meet some set of "utility" goals over the long term. Clearly both are necessary. The current state of the art is that neither discipline has a good solution to its "minor" component. Merging these two technology threads would be advantageous to both.

It also appears that much of our survivability modeling based on threat models could be applied to other areas than software survivability. For example, logistics planning software could be made more survivable, but the logistics plans themselves could be made more survivable by the application of some of the threat model based techniques we developed.

We are still pursuing funding to continue this work, particularly its insertion into the DARPA Information Assurance program. Continuation of this work is critical to the

success of componentware and is not being addressed elsewhere: neither the Java nor OMG communities are doing this kind of work.

Appendix A-1

Survivability in Object Services Architectures

David L. Wells, David E. Langworthy, Thomas J. Bannon
Object Services and Consulting, Inc.
Dallas, TX
{wells, del, bannon}@objs.com
September 1997

Abstract

This report describes the goals, approach, and anticipated results of the project "Survivability in Object Services Architectures". It also introduces a collection of other reports produced on the project.

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

Appendix A-1

Table of Contents

Appendix A-1

1. Introduction

As mission critical software applications become larger and increasingly geographically distributed, the frequency of loss or degradation of parts of the system due to physical or information warfare attacks, hardware or infrastructure failures, or software errors increases dramatically. At the same time, the complexity of such systems and the need to rapidly adapt them to changing real-world requirements renders inadequate many of the traditional means of making software robust.

This report gives a technical overview of a DARPA funded, Rome Laboratory administered project entitled *Survivability in Object Services Architectures* being executed by Object Services and Consulting, Inc., that is developing software models and mechanisms to address this problem. As of September, 1997, the project is about one year old, with a bit over one more year to run.

The paper is organized as follows. Sections 2-3 summarize the project's goals and anticipated results. Section 4 summarizes our technical approach. Section 5 lists other documents containing additional technical detail.

2. Project Goals

This project has three related top-level goals:

- to make military and commercial software applications based on the popular Object Services Architecture (OSA) model far more able to survive failure and attack than is currently possible,
- to make the development and use of survivable OSA-based applications tractable and cost-effective, and
- to scale to collections of numerous, large, independently developed applications running in the same computing and networking environment.

The second and third goals complement the first. If the development of survivable applications is too difficult, they will never be built, rendering moot the power of the survivability mechanisms. The need to scale is obvious. The need to address independently developed applications competing for resources reflects the reality of a world in which dedicated resources and tightly coupled, closed systems are rapidly giving way to shared resources and loosely coupled systems, often constructed at least partially from commercial off-the-shelf (COTS) or pre-existing (GOTS) software.

We restrict our attention to OSAs because unlike more general ways to construct applications, the OSA model and implementation (collections of object services interacting locally or remotely over an object bus) are clean enough that extensions are tractable. At the same time, OSAs are very powerful and are used increasingly within DOD and commercially. The two best known examples of OSAs are the Object Management Group (OMG) Common Object Request Broker (CORBA) and Microsoft Active-X.

Our three goals are discussed further below.

Appendix A-1

Survivable OSA Applications and Services: Robust applications must be able to survive software, hardware, and network failures and degradation. For applications to survive, it must be possible to reconfigure them as resources fail. When possible, the reconfiguration should maintain complete functionality and performance, but if the resource loss becomes too severe it must be possible to gracefully degrade the application(s)' capabilities to make best use of the remaining resources. Reorganization must be situational since different real-world situations place different valuations on application functionality. This requires mediating between conflicting demands for resources as the resource pool diminishes. Naturally, the survivability mechanisms themselves must be stable and not introduce additional points of weakness.

Cost Effective Use and Development: To make the development of survivable OSA-based applications tractable and cost-effective, our solution must reuse or adapt key existing software infrastructure, keep development simple, and be widely applicable. The software infrastructure to be reused comes primarily from two main areas: the OSA domain itself (development tools, ORBs, object repositories, and object services), and the combined domains of fault tolerance, high availability services, failure/attack detectors, and system monitors. The straightforwardness of OSA application development is largely responsible for the popularity of the model and must be preserved by our solution; i.e., development should remain approximately the same as it is now and complex specifications or nonstandard development tools should not be required. To achieve this, we make survivability orthogonal to conventional OSA application semantics; in other words survivability is "added" to an application rather than built into it from the start[1]. To ensure that the solution is widely applicable, we plan to make our specifications and prototype survivability tools publicly available, and work through the Object Management Group (OMG) to place our specifications and prototypes into their standards process.

Scaling and Independence of Development: Individual applications or services should not be responsible for the details of ensuring their own survivability since this is generally hard to program, does not amortize the cost of developing the survivability mechanisms, may conflict with other applications' or services' needs, and requires a more accurate knowledge of the eventual deployment environment(s) than is reasonable to expect at development time. This argues for survivability being provided by a "Survivability Service" that handles the survivability needs of applications collectively, responding to changes in workload, resource requirements, resource availability, and threats based on a number of models that can be specified independently. The models can be at various levels of fidelity, with higher fidelity resulting in better ability to survive. If an application's or service's requirements are not specified, the application/service will be unaffected by the existence of the Survivability Service, satisfying the goal of minimal impact. This approach supports the goal of development simplicity. A consequence of making survivability orthogonal to application functionality is that changing the models

---

[1] While this may sound unrealistic, examples of adding behavior to applications with few modifications exist, notably object-oriented databases which add persistence orthogonally to programming language structures.

(not the applications or services) allows applications to be deployed into dynamically changing or unanticipated environments.

3. Expected Results

Results of several kinds are expected from this project. In an effort of this size, a complete solution to such a complex problem, supported by robust software, is not possible. We have chosen to concentrate on providing the following types of results:

- **Models.** The key to constructing survivable systems is to configure them in such a way that they can be easily reconfigured when needed to survive loss of system resources. We are extending and clarifying the standard OSA model to define "survivable configurations" as ones that are able to withstand component loss and are also capable of being systematically evolved into new configurations should component loss become severe. The models specify how to change both the physical configuration (different service placement or resource allocation) and the logical configuration (service alternatives or changed levels of service quality).
- **Architecture Specification.** We are developing a specification for the architecture of a Survivability Service that implements the models described above. The architecture is compatible with existing OSAs and projected trends and encompasses a wide variety of existing research in fault tolerant systems, failure detectors, system models, etc. We concentrated initially on providing an overall architecture for the Survivability Service that covers the "big picture" of how the components relate. This includes an internal partitioning that allows major subsystems to be replaced or refined, possibly by third parties. Of considerably less importance, at least in the early stages, will be detailed interface descriptions (both the Survivability Service's API and its internal interfaces) since changes to these will have limited scope and are likely to evolve as development progresses and other development projects are integrated. We also plan to convert the specification to standard Object Management Group (OMG) format and submit it to the OMG as a draft specification or input to a Request for Information (RFI).
- **Prototype Software.** We are prototyping the parts of the Survivability Service related to decision making, including a market mechanism for resource allocation, simple models and model evolution to drive survivability decisions under changing conditions, specifications of how to rebind logically equivalent or similar services, and some visualization. This will allow demonstration of a cohesive part of the Survivability Service that can later be attached to failure detectors and actual ORBs to carry out decisions made in the part of the system implemented. This strategy also matches well with our understanding of work done by other projects so that we can avoid too much development overlap.

4. Technical Approach

OSA survivability is not a *tabula rasa*; there is already substantial existing work in a number of areas that bear directly on the problem, but that work is largely disjunct and does not solve all of the problems. Our approach is to identify useful existing and

proposed technology (largely from the research community, since little of this technology has penetrated the commercial world), determine how this can be applied to OSAs, and integrate it into a unified whole. The result is a layered architecture for a Survivability Service that provides increasingly sophisticated kinds of survival strategies using a market mechanisms to allocate resources based on a number of models. Survivability is added to applications and services by exploiting properties of OSAs that allow the Survivability Service to seize control when needed to reconfigure the system. The major aspects of the system are outlined below.

4.1. Survivability Architecture

Survivability requires a variety of actions that are organized in the Survivability Service into the following layers.

- **Basic Process Control:** The ability to start, stop and restart processes, to clean up after failed or aborted processes, and to restore processes to known states. Most of this is provided by ORBs.
- **Fault Tolerant Services:** These are services designed to (usually) fail in known "good" ways. Their failure modes become part of the service specification. This must be provided by the service developers.
- **Failure Detection & Classification:** These mechanisms detect the symptoms of failures and attacks, and classify the events into likely failure categories. This can be done through probes, wrappers, or exception reports from well-behaved services. Classifying observed symptoms into error categories is at least partially based on the failure mode specifications of the fault tolerant services. We are not working in this area, and will either obtain these mechanisms from elsewhere or assume an oracle for demonstration purposes.
- **High Service Availability:** These are a collection of mechanisms to make individual service instances much more highly available than they would otherwise be. Techniques are either based on replication or hierarchical masking (i.e., error handling in the client). We concentrate on using replication-based policies since they do not rely on the semantics of the services and are therefore more widely applicable. Many replication-based policies exist (e.g., voting, hot backup, error correction) and some are integrated with ORBs (e.g., Electra and Orbix+Isis). These mechanisms must be efficient since they are invoked during normal (non-error) operation. At this level, it becomes possible to physically reconfigure an application by changing the way individual services are implemented. The logical organization remains fixed in that clients still interact with the same services after any reconfiguration.
- **Availability Management:** This layer manages the use of the High Service Availability mechanisms. It determines the appropriate fault tolerance mechanism to use for a given service based on service failure modes and perceived threats, and determines the resource pool needed to achieve desired availability. It can be less efficient than the lower layers since its use is infrequent or can be a background activity.

- **Service Renegotiation:** At this level, it becomes possible to change the logical organization of an application by binding clients to alternate services if the desired service should become unavailable or degrades in performance. The rebinding can be to an equivalent, but distinct service (e.g., a different server having the same maps), or to a similar, but acceptable service (e.g., a different server with maps of the same area but at lower resolution). Alternatively, the same service connection can be maintained but at a lower quality of service (e.g., more errors or slower). This is semantically more sophisticated than lower layers and requires specifications of client-service connections beyond those currently used in OSAs. Use of threat, situation, and resource models is definitely recommended. In addition to allowing rebinding to service alternatives when services fail, service renegotiation can represent a fallback position if the costs of assuring service availability become unacceptably high.

4.2. Models

A variety of models are used by the Survivability Service. All models must be distributed. The models include:

- **Resource Model:** This captures physical resources, services, and code that implement the various services. It will definitely be partitioned.
- **System Model:** This defines the perceived current system configuration. It may be inaccurate, since it will be asynchronously updated and failures may not be detected until some time after they occur.
- **Attack/Failure Model:** This defines the types of possible attacks and failures, and the consequences (affected resources) of each.
- **Threat Model:** This augments the Attack/Failure Model by adding the anticipated likelihood of each kind of attack or failure. It is used when classifying errors and when determining how to reallocate resources (it does little good to rely on a resource believed to be under increased attack). The Threat Model may be influenced by the Situation Model.
- **Situation Model:** This defines the relative importance of tasks in the current real-world situation and modifies the Threat Model according to threats that are situation-based (e.g., physical attack is more likely when at war).

4.3. OSA Hooks

The various survivability techniques discussed above must be integrated into the OSA framework. This is straightforward, since one of the main characteristics of OSAs is the loose, well-defined boundary between clients and services. While not currently part of the OMG CORBA 2.0 specification, the ability to trap traffic across the ORB has been requested by the OMG Security Service and will probably be part of future CORBA specifications. This would allow binding to service replicas and rebinding to service alternatives in a straightforward fashion. Simple extensions (or protocols on top of) the

way in which services are launched would support the choices of implementation type and location essential to place services intelligently.

5. Project Documents

The following reports have been (or are expected to be) produced under this project. Because we are at a midpoint in the project, even those reports that are currently in "final" form are expected to be revised next year as we refine the models and specifications, and as development of the Survivability Service progresses.

- *Survivability in Object Services Architectures*. [this report] This report describes the goals, approach, and anticipated results of the project "Survivability in Object Services Architectures". It also introduces a collection of other reports produced on the project.
- *Composition Model for Object Services Architectures*. [9/97, revision expected 10/98] This report describes extensions to the standard Object Services Architecture model to support composition of OSA-based applications from object services using external binding specifications. Isolating the decision about which particular service to bind to from the abstract specification of the characteristics of the service required allows binding decisions to be reasoned about in the context of global system knowledge generally unavailable to the developer of an individual application, either because the environment is too complex to be fully understood, because the environment is changing dynamically as the result of attacks or failures, or because the system is being deployed in an unanticipated environment. This gives the ability to tailor application configuration based on current resource utilization and perceived threats to the system resources. The result is the ability to configure more survivable OSA-based applications than would otherwise be possible. The OSA Composition Model is the basis for the *Evolution Model for OSAs* and supporting *Evolution Tools for OSAs* which migrate application configurations from one legitimate state to another.
- *Evolution Model for Object Services Architectures*. [9/97, revision expected 10/98] This report describes extensions to the Object Services Architecture model that make it possible to safely migrate a running application from one legitimate configuration into another legitimate configuration. Both semantically identical and semantically similar transformations are possible under this model, which allows applications to continue to survive in degraded mode when system resources become unavailable due to attack or failure. Legitimate transformations are determined based on the original application service binding specifications as described in the *Composition Model for OSAs* and mapping rules that define various possible transformations. From within the set of legal evolution possibilities, a number of system and threat models are used to determine a "good" transformation based on a malleable combination of predicted safety, best performance, and lowest cost.
- *Evolution Support Toolset for Object Services Architectures*. [to appear 11/97, revision expected 11/98] This report describes the architecture of an OSA Survivability Service that uses the *OSA Composition Model* to initially configure

Appendix A-1

OSA-based applications and reconfigures them for survivability using the *OSA Evolution Model*. The Survivability Service uses a single set of system models and specifications for both purposes. The Survivability Service is compatible with existing work in failure detection and classification, fault tolerance, and highly available systems. Both the internal architecture of the Survivability Service and its connections to external services are described. Portions of the Survivability Service are being prototyped as part of this project.

- *User Manual for the Evolution Toolset for Object Services Architectures*. [to appear 11/98]. User and installation guide for the Evolution Toolset prototype. Lists limitations and known bugs.
- *OMG Object Change Management Service Proposal*. [to appear 10/98] A proposal to the Object management Group for a Change Management Service based on the work performed on this project. The report will be basically a rewrite of the other project documents into a form compatible with OMG standards for draft specifications.

Appendix A-2

Composition Model for Object Services Architectures

David L. Wells, David E. Langworthy,
Thomas J. Bannon, Nancy E. Wells, Venu Vasudevan,
Object Services and Consulting, Inc.
Dallas, TX
{wells, del, bannon, nwells, venu}@objs.com

Abstract

This report describes extensions to the standard Object Services Architecture model. The extensions support the composition of OSA-based applications from object services using external binding specifications, making it possible to isolate the decision about which particular service to bind from the abstract specification of the characteristics of the service required. This allows binding decisions to be reasoned about in the context of global system knowledge generally unavailable to the developer of an individual application, either because the environment is too complex to be fully understood, because the environment is changing dynamically as the result of attacks or failures, or because the system is being deployed in an unanticipated environment. This gives the ability to tailor application configuration based on current resource utilization and perceived threats to the system resources. The result is the ability to configure more survivable OSA-based applications than would otherwise be possible. The OSA Composition Model is used by the *Evolution Model for OSAs* and supporting *Evolution Tools for OSAs* which migrate application configurations from one legitimate state to another.

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

Appendix A-2

Table of Contents

Appendix A-2

1. Introduction

An OSA-based application consists of a collection of object services interacting across an object bus. Not all possible configurations of OSA-based applications are equally robust, nor are all configurations equally able to be reconfigured. To facilitate the construction and maintenance of survivable OSA-based applications, it is desirable to define a subset of the possible OSA configurations that is known to be more survivable and that can be reconfigured into another configuration in the survivable set. Such configurations should be easy to construct. The *OSA Composition Model* described in this paper defines such a subset of OSAs.

The paper is organized as follows. Section 2 gives an overview of OSAs and OSA-based applications, and identifies weaknesses in current technology that cause systems to by non-survivable. Section 3 discusses variations in the level of abstraction at which objects and OSAs can be defined. This is important, since our approach is to define higher levels of abstractions that allow the creation of survivable configurations. These higher levels of abstraction, as well as a number of models needed to support them, are defined in Section 4. Section 5 discusses the properties of existing OSA implementations onto which our abstraction is mapped; Section 6 describes this mapping. Section 7 gives the current (working) definitions of the various models introduced in Section 4.

2. Overview of OSAs and OSA-Based Applications

An OSA-based application consists of a collection of *objects* interacting across an *object bus*. A calling object is called a *client*; a called object is called a *service*. An object may be both a client and a service.

Every object is characterized by a unique *OID,* one or more *interfaces* defining the *services* the object can provide, and *state* reflecting the effects of past operations performed on the object through its interfaces. An object's interface is specified by an *interface type* that defines the collection of *methods* by which objects of that type can be manipulated. Interfaces may be related by subtyping, but this is not necessary; i.e., an object may provide the services defined by two interface types without the existence of an interface type derived from either.

The object bus over which objects interact is a messaging system by which services are located, instantiated (if necessary) and bound to clients, methods are invoked, and results and exceptions returned. The bus may be synchronous or asynchronous. Broadcast and multi-cast may or may not be supported in addition to point-cast. Objects interacting across the bus may reside in the same process, a different process on the same machine, or on different machines. Even if the communicating objects reside in the same process (address space), the existence of the object bus between them is important since it provides a control point where mediation can take place transparent to either object.

In order to perform the methods defined by an object interface, the interface type must be implemented in code. An *implementation class* implements the behavior of the type and provides an object bus connection. There may be multiple implementation classes for a given interface type. This allows alternate implementations with different time/space tradeoffs, different environments in which they can run, different languages, etc. A given

implementation class can implement multiple interface types; this is often exploited when existing code is "objectized" by wrapping.

In order to be operated upon, a service must be *instantiated.* A *service instantiation* consists of an OID, the code for an implementation class, and state in a form compatible with the instantiation class. While logically an object's OID, state, interface, and implementation are bundled, in practice they are separate and are managed differently (generally, a single copy of the implementation is used for many objects, and state is frequently stored separately in a DBMS or file system). The object bus generally is able to instantiate a service (including assembling the constituent parts and launching it) if an attempt is made to use a service that is not already instantiated. This prevents resources from being consumed by services that are not currently being used.

Because clients and services interact across the object bus, the client cannot directly manipulate the service instantiation. Method invocation and result return is done through a *surrogate* that resides with the client and provides a connection to the service instantiation. A surrogate (often called a proxy) is implemented by a *surrogate class*, which, like the implementation class, is a subtype of the server object's interface type. Like implementation classes, there may be multiple surrogate classes for a given interface type. This allows surrogates to be in different languages or to be customized to provide additional functionality (e.g., caching).

To actually cause information to be transmitted across the object bus, both the surrogate class and the implementation class must package and unpackage arguments and return values (this is called *marshaling* and *unmarshaling*). Code to perform these tasks can be generated from the interface type, so it places no burden on the service implementer.

For a client to use a service, the service must be *bound.* In binding, a service instance is identified and a surrogate for that service instance is created in the client process. The service instance being bound is ultimately identified by its OID. The determination of which service instance to bind can be done through a complex process if supported by the OSA; increasing the sophistication and flexibility of this binding process is one of the keys to OSA survivability.

3. Levels of Abstraction

In the above description of OSAs and OSA-based applications, only a single level of abstraction is presented. However, this oversimplification of reality confuses a number of issues. We concern ourselves with abstractions of both objects and architectures. These are discussed in the following two subsections.

3.1 Object Abstraction

Objects are defined and implemented at a number of levels of abstraction. While all of the object concepts such as OID, interface, implementation, and state exist at all levels, their meanings and the ways in which they can be used vary with the level. For this reason, when defining or using a construct, we must be careful to be clear which abstraction we are dealing with.

Appendix A-2

3.1.1 Examples

Consider the following related motivating examples based on the concepts of object identity and instantiation. Two fundamental properties apply to the object abstraction at any given level:

- all operations performed against an object contribute to the object's abstract state, which in turn affects all subsequent users of the object, regardless of how they access it, and
- all object handles to the same service must be equivalent (i.e., return "true" to some appropriate comparison function).

Note that this says nothing at all about how these simple conditions are implemented.

An obvious way to enforce the conditions is by having a single instantiation of the object, which is reached via object references related to the object's OID. However, this does not work if the object's concrete state (a particular representation of the abstract state that is manipulable by the implementation) must exist in more than one address space simultaneously. This is the case with persistent objects, which must exist in memory to be manipulated and must exist on disk (or similar) in order to persist. A DBMS keeps the two instantiations consistent at appropriate times (fault, commit, etc.) and disposes of the memory instantiation when no longer needed. The client always receives handles to the memory instantiation, so it appears that there is only one instantiation and one OID, but the DBMS must be aware of both instantiations, each with its own, distinct, OID. Thus, the object at the higher level of abstraction is implemented by two objects at the lower level of abstraction. Note that this is not the same as a composite object at the higher level of abstraction, in which multiple higher level objects are accessed via a single higher level object. In our example, the lower level objects have no existence at the higher level of abstraction.

Another reason for different levels of abstraction is exemplified by early implementations of LISP. Everything, including numbers, was reached via pointers that behave like OIDs in that they uniquely identify things. The result was that a program had a single instantiation of "5", pointed to from multiple places. This was inefficient in space (the pointer is the same size as the integer it points to) and time (pointer following consumes an instruction). An obvious optimization was to embed the number back into the position occupied by the pointer and operate on it directly. This resulted in multiple instantiations of "5". However, despite having multiple instantiations of the same "proto-object", the consistency constraint was not violated because the "5" is immutable. Since "5" was never modified, the fact that there were multiple instantiations didn't matter, since all would have the same abstract state. Better still, no effort was required to keep them consistent. Here again, the abstraction of a single "5" was not the reality at a lower level of abstraction. Handle equivalence means simply redefining the equivalence operator to return true if objects of that type have the same value.

The above example has an analogy in OSAs. When objects become widely distributed or heavily used, there are significant advantages to be gained from replicating them to reduce bottlenecks or communications costs. Of course, unless the objects are

immutable, coordination must occur among the replicas to ensure that they behave consistently. It is important that the client of the object use a single OID regardless which replica is used, since otherwise comparisons of object handles will not be consistent. However, the mechanism that coordinates the replicas must differentiate amongst them, which requires different handles and hence different OIDs. Note that this applies to implementation classes and interface types as well as to object instantiations.

Each of the four key aspects of objects: OID, interface, state, and implementation may be affected at the various levels of abstraction typically found. Below is a rough summary of the kinds of changes seen in each as the level of abstraction changes. There is no attempt to be complete or to burrow down to the lowest level of abstraction, since that is ultimately just bits.

The levels described here are not absolute, and any given object model may mix the properties from various layers. However, it is unlikely that any one abstraction will be substantially more sophisticated than the others, because of the obvious linkages between them. Naturally, there is a tendency to become more machine-oriented as the level of abstraction gets lower. This does not necessarily mean less complex.

3.1.2 OIDs

At low levels of object abstraction, OIDs are simple; they tend to identify specific physical locations; e.g., a disk or memory location, port address, etc.). These OIDs also tend to have limited scope; e.g., addresses in a single machine or subnet.

At intermediate levels of object abstraction, OIDs tend to have a more complex internal structure that is used to encode properties of the object such as its type and where its instantiation or persistent state resides. This makes for greater efficiency when following the OID to the instantiation, but also introduces some rigidity since these properties of the object that should be able to be changed transparently to the client cannot be changed without changing the OID.

At high levels of object abstraction, OIDs are again simple, but now they are very abstract. There tends to be little or no substructure to a such OIDs; what structure there is is usually a consequence of partitioning the OID domain among a number of OID allocation services to avoid bottlenecks when new objects are created. If OIDs are allocated on a per type basis, the OID may imply an object's type, although this is undesirable since it precludes evolving the object's type. Such OIDs have global scope; an OID is unique across the entire span of the OSA.

3.1.3 Interface

At low levels of object abstraction, interface is intimately tied to the programming language of the implementation. Subtyping is allowed. Types cannot evolve.

At intermediate levels of object abstraction, interface types become language independent.

At higher levels of object abstraction, types may evolve while maintaining the same type identity. Objects may evolve with the type or versions may be maintained.

At still higher levels of abstraction, an object's type may change either by being refined or changed altogether. For example, a vehicle is determined to be a truck, then refined to be a Ford, then the classification is determined to have been wrong, so the vehicle is reclassified to be a Chevrolet.

3.1.4 State

At lower levels of object abstraction, state is defined in terms of the programming language of the implementation.

At intermediate levels of object abstraction, multiple copies of the state may exist (e.g., persistent and manipulable), but only the representation used by an object instantiation may be manipulated from a client. The others exist solely for housekeeping within the object layer, such as allowing an object to retain its state even when not instantiated.

At higher levels of object abstraction, object state must be able to be transferred from one representation to another to support multiple implementation classes for an instantiation and to support type (and hence class) evolution.

3.1.5 Implementation

At low levels of object abstraction, implementation is language-specific. Polymorphism allows an interface type to have multiple implementation classes, but any individual object will have only a single instantiation type. Instantiations are limited to a single address space (note this does not preclude the implementation from distributing itself, but there is no support in the object abstraction for this). An instantiation consists of a single copy of an implementation class; there is no support for replication.

At intermediate levels of object abstraction, the implementation classes of a type may be from different programming languages. A service need not be in the same language as a client calling it. Little else changes.

At high levels of object abstraction, there is a great deal of implementation independence. An individual object can have multiple implementations over time. This allows bug fixes, performance improvements, and object migration by allowing the object's instantiation to move to different platforms and platform types. An object's implementation can be replicated, with the replication being supported by the abstraction.

3.2 Architectural Levels

Typically in OSA-based applications, one finds two levels of architecture: an *application architecture* and an *OSA implementation architecture.* Any architecture defines the (potential) functionality, configuration, and multiplicity of the components of systems instantiated under that architecture. In other words, it defines the properties that must be met by any *legitimate* instantiation. The notion of *legitimacy*, expanded on below, is key.

An *application architecture* defines these properties for a particular application (e.g., avionics for the F-22 with specific on-board sensors) or application family (e.g., avionics for a class of aircraft with a variety of sensor types). It is obviously specific to a particular application domain.

Appendix A-2

The *OSA implementation architecture* defines an object model on which applications are built. It corresponds closely (or exactly) to what is provided by an OSA implementation such as CORBA, Active-X or Java. An OSA implementation architecture is non-application-specific, but corresponds closely to the details of the object model and messaging implementation of the object bus that will be used. As such, it is closely tied to existing technology.

To make the construction of survivable OSA-based applications tractable, we argue that two levels of abstraction is insufficient. A third layer, which we call the *OSA survivability architecture*, is needed between the other two layers. To justify this, consider these (necessary) limitations of the application and OSA implementation architectures.

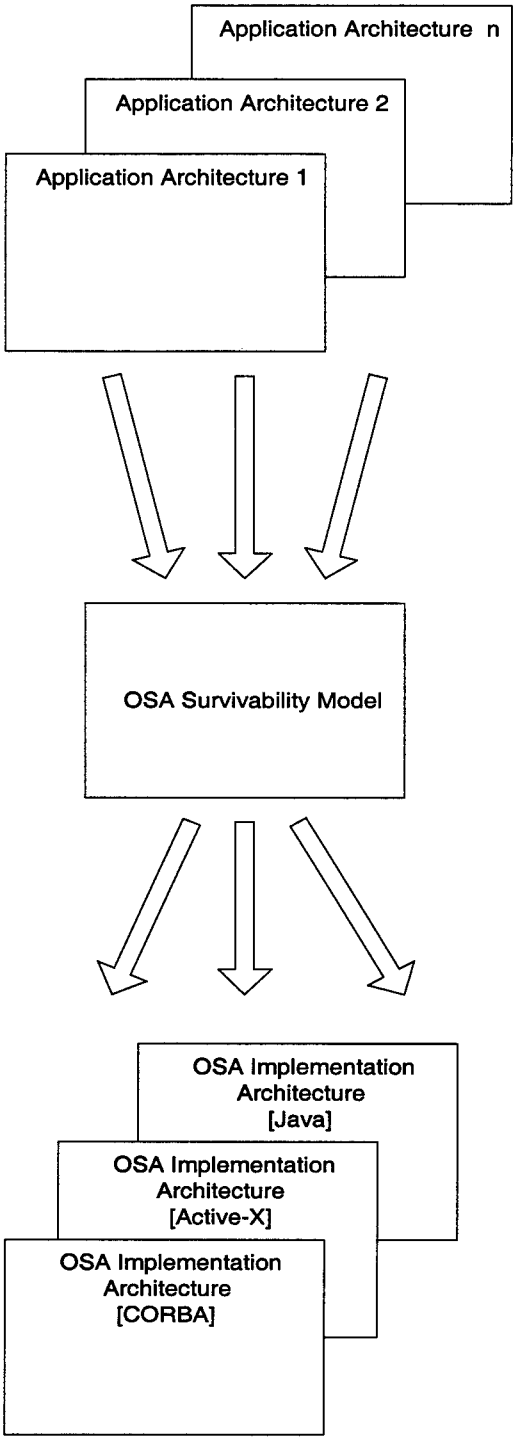
The OSA implementation architecture defines a minimal set of necessary properties. Mechanisms are defined to specify interface types and to associate implementation classes with them. Objects can be created and destroyed. They can also be registered with the object bus so that they can be subsequently located, bound, sent messages, and return results or exceptions. Only those properties necessary to perform those operations are defined. There are two rationales for staying simple at this level: the model must be implemented efficiently on a variety of platforms and languages, and a generic object model can support a wider variety of uses than can a more specific model. Together, this results in an object model roughly equivalent to what was defined above as an intermediate level of object abstraction.

One drawback to this approach as the level on which applications are directly constructed is that features such as replication, type evolution, etc., that can be provided in higher level object abstractions are not provided. Thus, they will either be foregone, or will be developed in an idiosyncratic way that is time consuming and will not interoperate with other idiosyncratic solutions to the same problem. Another drawback is that such a general architecture admits a large number of undesirable configurations as legitimate. Some configurations are simply inefficient or brittle; e.g., placement of all replicas of an object on the same host machine. Some are syntactically correct but semantically meaningless; e.g., binding to a map service with the right interface type but that contains maps of the wrong region. Finally, forcing the application to deal with these issues means that they will be decided at development time, or at best, when a specific instantiation is deployed; the underlying system can give no support for evolving an application to meet survivability goals.

The application architecture, as observed above, also is an inappropriate place to address these limitations, since the solution to them is applicable to many application domains, and development cost should therefore be amortized. Equally important, to make a collection of systems sharing a resource pool able to collectively survive based on global notions of requirements and worth, it is essential that no application be solely responsible for its own survival, since this precludes systemwide decisions from being made.

Multiple application architectures use the OSA survivability model, which in turn can reside on top of a number of OSA implementation architectures. This is shown in the figure below.

Appendix A-2



3.3 Goals of the OSA Survivability Model

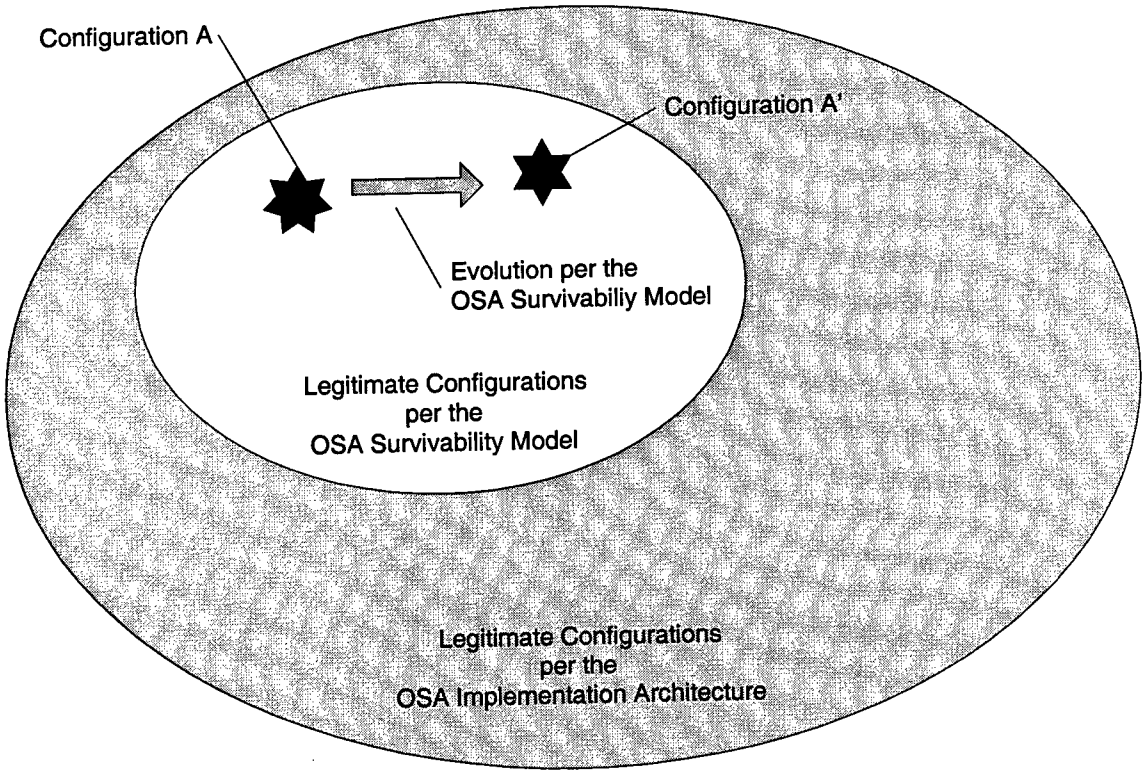
The OSA survivability architecture is intended to accomplish the following:

- provide a *higher level object model* that frees application developers from the rigidities of the OSA implementation architectures currently extant,

- restrict the set of legitimate configurations admitted by the OSA implementation architecture to a subset of *survivable configurations* that are themselves robust and that can further be evolved into other survivable configurations (evolution is discussed in *Evolution Model for Object Services Architectures*),
- provide *places to introduce the survivability mechanisms* of basic process control, fault tolerant services, failure detection and classification, high service availability, availability management, and service renegotiation as provided by the Survivability Service[1], and
- be mappable to a variety of OSA implementation architectures, notably CORBA, Active-X, and Java (evolutions must also be mappable).

The OSA Survivability Model is composed of two submodels: an OSA Composition Model (described in this report) that defines legitimate configurations, and an OSA Evolution Model (described in *Evolution Model for Object Services Architectures*) that defines moves between legitimate configurations. Naturally, the two models are closely related, since we avoid putting a construct into the Composition Model unless we know how to evolve it. This means that it is likely that both models will gain additional constructs over time as we discover new evolution strategies.

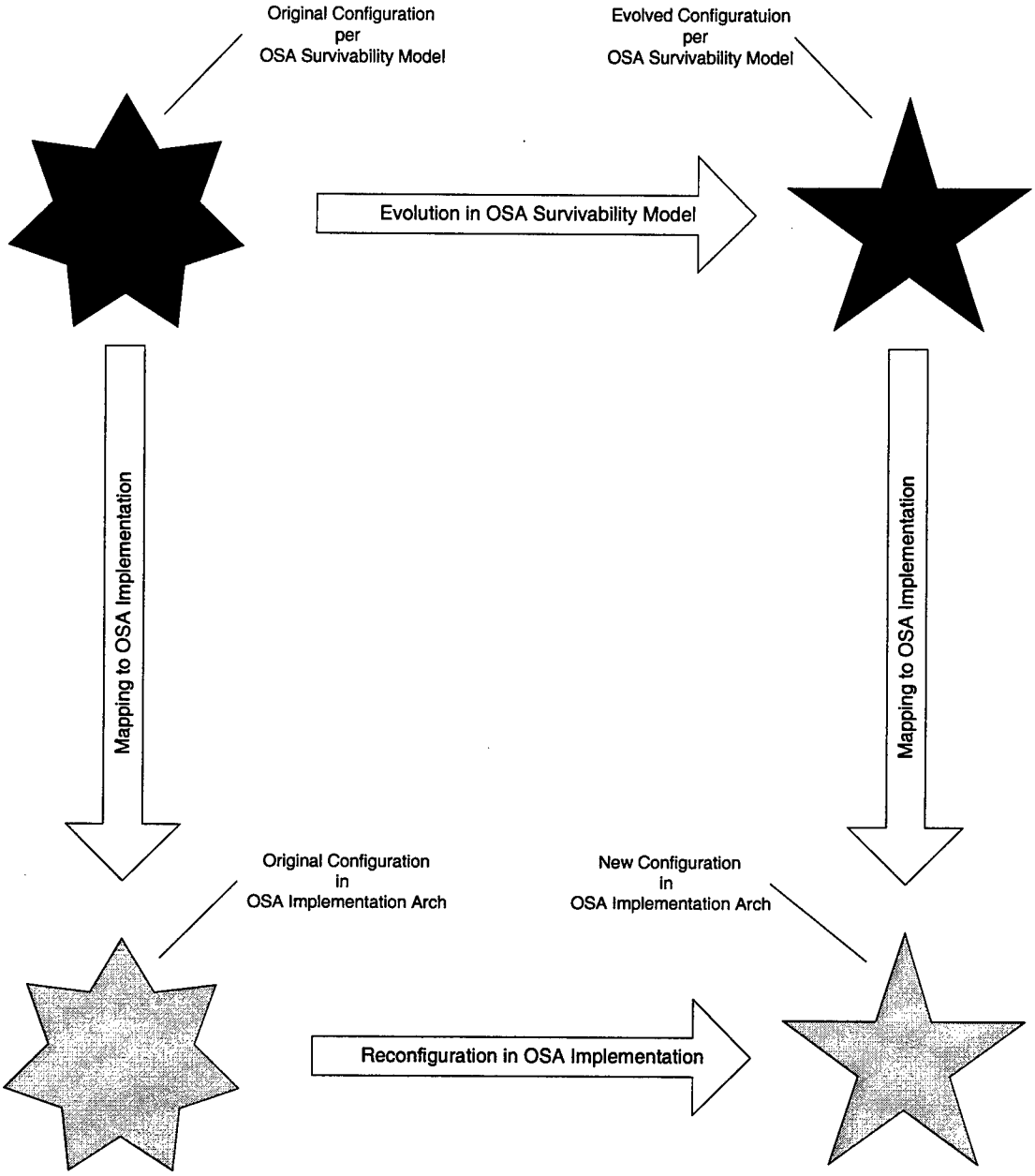
Restriction of legitimate configurations is shown in the figure below.



[1] See *Survivability in Object Services Architectures* for a brief overview of these mechanisms.

Appendix A-2

Mappings to the OSA implementation architecture and evolution at both levels are shown in the following figure. Note that while evolution takes place within the OSA Survivability Model, this corresponds to reconfiguration at the OSA implementation level. This is because the OSA survivability model contains the concept of evolution, while the implementation level does not. The result is that the new and old configurations are related only at the survivability level; at the implementation level, the two configurations are both legitimate according to the OSA composition model, but are otherwise unrelated (at that level of abstraction).



Appendix A-2

The OSA Composition Model and OSA Evolution Model specify *desirable* configurations and moves as well as *legitimate* ones. Obviously, not all legitimate configurations are equally desirable in one sense or another. For example, a configuration that places all replicas on a single machine in a burning building would not be especially desirable from a survivability standpoint, although the momentary performance might be excellent because of lack of competition for the resources. Also, some configurations may be more desirable because they offer a higher QoS than some other configuration that is also acceptable.

4. OSA Composition Model

To support the construction of survivable configurations that are not directly created by application developers, the Composition Model consists of:

- a *clean, high level object abstraction* in which developers specify, implement, and connect services. The object abstraction:
  - makes a very clean distinction between the abstraction of a service instance and its implementation(s) in order to support replication, instance migration, change of implementation class for a given service instance, and multiple simultaneous implementation classes for a given instance;
  - abstracts the bindings of clients to services and implementations to resources in order to allow an OSA Survivability Service to determine which service instance best meets the needs of a client, and how and where that service instance should be instantiated;
  - defines useful patterns of object configurations that have desirable survivability properties that can be instantiated by the Survivability Service; and
  - uses the concept of *quality of service (QoS)* to allow alternatives to both service bindings and implementation instantiations in the event resource limitations prevent optimal behavior.
- *resource, attack/failure, threat, and situation models* that collectively are used to determine:
  - which of the acceptable configurations specified by the developer using the object abstraction are actually instantiable given the available resources; and
  - which of the instantiable configurations is the most desirable given the competing demands for resources, the values of the various services, and perceived threats to resources.

The remainder of this section presents our object abstraction and the supporting models. During this discussion, we identify how the various survivability mechanisms (basic process control, fault tolerant services, failure detection and classification, high service availability, and availability management) fit into the abstraction and models. Service renegotiation is discussed at length in *Evolution Model for Object Services Architectures*.

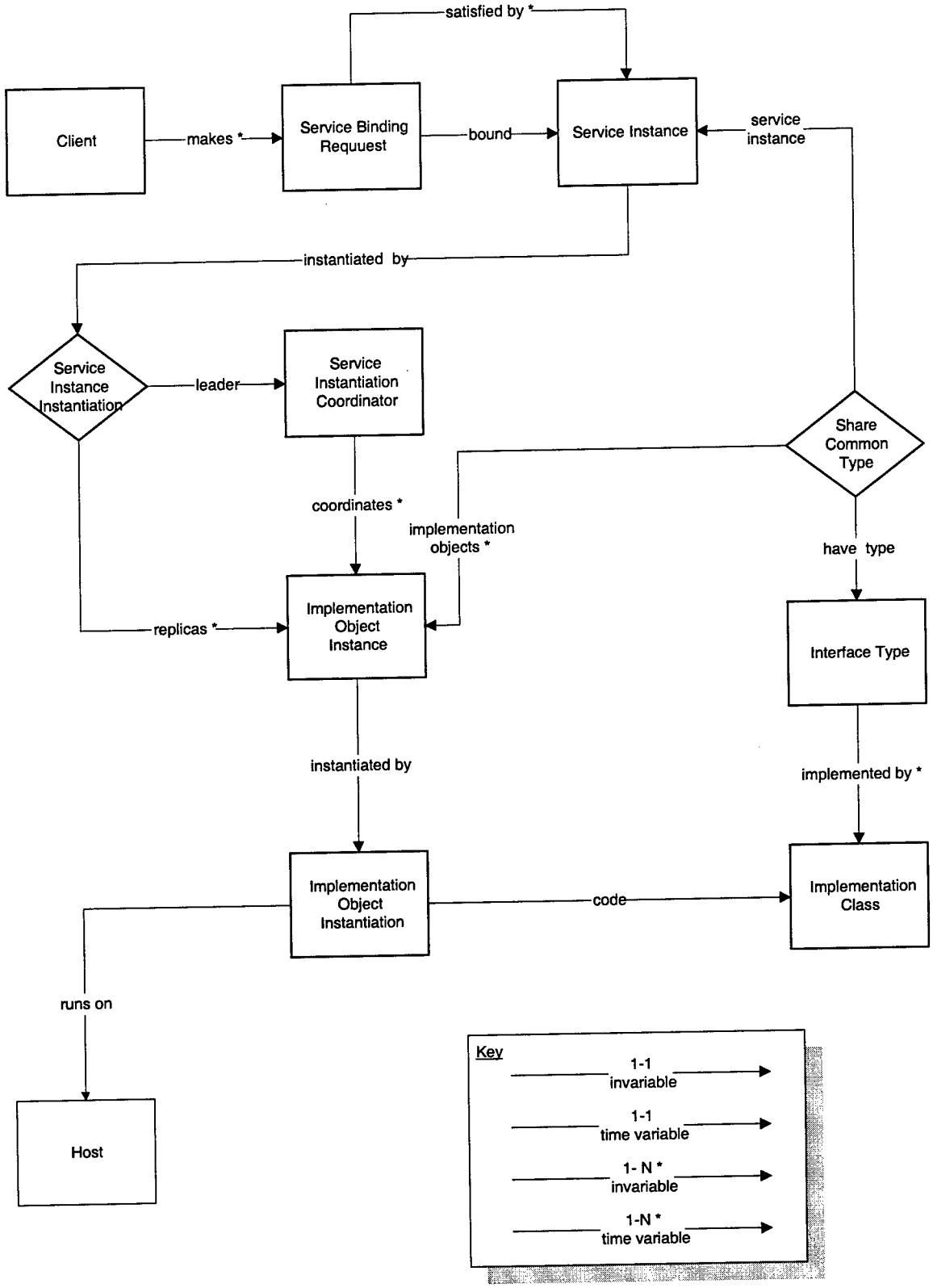
Appendix A-2

4.1 Object Abstraction

We base our abstraction (and terminology) to the largest extent possible on the OMG object and object bus models, clarifying and adding new features where necessary for the goals of survivability. Active-X is very similar to the OMG models in the important dimensions, so similar extensions and mappings there appear straightforward. Java has different strengths and weaknesses, but extensions and mappings also appear tractable in that domain.

The OSA Composition Model's object abstraction is illustrated by the figure below.

Appendix A-2


satisfied by *
Client
makes *
Service Binding Requuest
bound
Service Instance
service instance
instantiated by
Service Instance Instantiation
leader
Service Instantiation Coordinator
Share Common Type
coordinates *
implementation objects *
have type
replicas *
Implementation Object Instance
Interface Type
instantiated by
implemented by *
Implementation Object Instantiation
code
Implementation Class
runs on
Host
Key
1-1 invariable
1-1 time variable
1- N * invariable
1-N * time variable

Appendix A-2

Objects are defined at three levels of abstraction (from high to low): *services* and *service instances, service instantiations,* and *implementation object instances* and *implementation object instantiations.* Connections between objects are always specified at the level of services and service instances. The functionality (i.e., code) of services is implemented at the level of implementation object instances and implementation object instantiations, as is their instantiation as CORBA objects. Service instantiations, known only to the Survivability Service, form the bridge between the other two levels.

In the abstraction, *clients* make requests for *services* by specifying the service required as a *service binding request*, which is a statement of what the requested service must do. For any given service request, there are potentially many *service instances* able to provide that service. One of these service instances is chosen to be *bound* to the client to actually provide the service. The pool of service instances that can satisfy the service request may vary over time, as may the choice of service instance to be actually bound[2].

In order for the service instance to be able to receive messages (necessary to actually do anything), it must be *instantiated*; the constructs to instantiate a service instance are collectively called a *service instance instantiation.* While there may be only one of these at a time for a given service instance, an instance need not always be instantiated in the same way. A service instance instantiation is composed of a *service instantiation coordinator* and one or more *implementation object instances.* As its name implies, the service instantiation coordinator coordinates the activities of the implementation objects. The implementation objects implement the functionality of the service instance. These are objects at the level of the implementation OSA, e.g., CORBA. As such, they are implemented (defined and coded) and instantiated (scheduled, executed, and set up to receive messages) as specified by the implementation OSA being used. For a given service instance instantiation, both the service instantiation coordinator and the collection of implementation object instances may vary over time. If not currently bound, a service instance need not be instantiated, allowing resources to be reclaimed.

Interface types and *implementation classes* are defined as usual, and there may be multiple implementation classes for a given interface type. Service instances and implementation object instances share a common interface type. This allows messages sent to a service instance to be forwarded to the underlying implementation object instances without reinterpretation.

The instantiations of service instances and implementation objects are fundamentally different in two ways. First, implementation objects are instances of some *implementation class* defining the data structures and code; this is specific to the semantics of the type. On the other hand, service instances are in no way associated with a particular implementation class; instead, they are implemented by the combination of a service instance coordinator and a collection of implementation objects. The service instantiation coordinator is not type-specific to the implementation object instances[3],

---

[2] Knowing alternatives to the bound service is essential to evolving the system should the initially bound service fail. This is discussed more in *Evolution Model for Object Services Architectures*.

[3] Except possibly as a convenience to cause signatures to match, but this can be generated automatically.

which means that a coordinator can coordinate for any type of implementation objects. Importantly, the implementation objects in the collection may change over time, meaning that while an individual implementation object is tied to a particular implementation class (and in CORBA to a particular host), no such restriction applies to service instances. The second major difference is that implementation object instances are instantiated in a single (as far as the ORB is concerned) process, while service instances may be instantiated in multiple processes on possibly different hosts. This has implications on the relative failure modes of service instance instantiations and implementation object instantiations that are discussed below. An implementation object may be instantiated in different locations each time it is instantiated; this may be directly supported by the Implementation ORB or may be added.

We now consider in somewhat more detail what this looks like to application developers, service implementers, and the Survivability Service.

From the perspective of a requester of computation, the world is organized as a collection of *service instances*, each of which performs a *service.* A *service* is defined very generally as useful work performed on behalf of a client by some entity outside the client. Services are provided by service instances; the concept of a service has meaning only to a client.

Services respond to messages sent by the client. Because it is often the case that the messages from a client requesting a service are related, a client is *bound* to a service instance to ensure that all messages reach the same service instance.

Every service instance is an object, and as such is characterized by an OID, one or more interface types (possibly related by subtyping) each of which defines a set of functions to which objects of that type respond, and an abstract state reflecting past operations performed on the object through its various interfaces. The service provided by a particular service instance is dependent on its interface (the kinds of things it does), its abstract state (what it knows), changes in state as the result of requests from other simultaneous clients (object identity determines this), and the quality of the results (in some QoS metric).

It is possible that at any given time, exactly the same service could be provided by more than one service instance. For instance, if two Map_Service instances (say one at the University of Texas and the other at Texas A&M) start out at 12PM managing identical maps, they provide the same service as long as QoS is the same and there is no interaction with other clients. If this could be ensured, either service instance would be acceptable to the client. Note that since these are distinct service instances, by 1PM updates could cause them to provide different services. For example, the A&M Map_Service instance could have been updated to record flood damage not recorded at the other service instance.

It is possible that non-identical services may be indistinguishable to the client requesting the service. In this case also either service instance would suffice.

A more interesting case is when differences exist between services, but these differences are irrelevant to the client. Consider again the pair of Map_Service instances. If a client will only request maps of Texas, the fact that the A&M Map_Service does not have the

same set of lunar maps as the University of Texas Map_Service instance is irrelevant. Many other examples can be considered related to QoS (both deliver the maps in an acceptable amount of time, though A&M is faster). Sometimes physical position of the service instance matters, as in the case of a print service that is tied to a particular printer; printing to either of two printers in the same room would probably be acceptable, but printing in Australia might not be.

It is because of the above that a client requests a service, not a service instance. This request is made as an abstract *service binding request* specifying what the client really needs. This specification is interpreted by the Survivability Service and mapped to a set (hopefully non-empty) of service instances that have the potential to satisfy the service request. Determining which service instances can satisfy a given service specification is non-trivial, and requires information about service instances that is not currently available in OSAs. Essentially, all service instances must advertise their (potential) properties that may be of interest to their likely clients. It is not clear at this point what properties are most needed, but minimally it must be possible to describe service type, abstract state, other clients to be interacted with (or not), and QoS. The vocabulary of terms to be used in service requests and service instance advertisements will likely be service-specific; at this point it is also not clear exactly what this vocabulary should be[4].

As a special case, it is important that type and OID be legitimate service requirement specifications, since this is a common, and efficient, way to bind. A service instance's OID is a degenerate way to specify both abstract state and interactions with other clients. However, since it does not admit a set of possible service instances, it gives away an important degree of freedom for survivability. Note that although CORBA provides some abstraction of the binding of a client to a service instance (or what in our terminology would be a service instance), it is limited to path completion, and does not concern abstract state or QoS. As we shall see shortly, this poses a different problem since it fixes the implementation of the service instance unacceptably, thereby giving away an even more important degree of freedom.

Besides providing the correct semantics, a service instance must be instantiable in order to potentially satisfy a binding request. There are a variety of reasons why a particular service instance might not be instantiable in a way that satisfies a service binding request. Among these are insufficient physical resources exist to deliver the required QoS, all possible implementations being deemed to be inaccessible or compromised (corrupted), the (persistent) concrete representation of the abstract state is unavailable, etc. A collection of models (described in section 4.2) are used by the Survivability Service to make this determination.

Note that the set of service instances that can satisfy a service binding requirement is potentially huge. It is not required that the entire set actually be computed. This is because only one service instance from the set will actually be bound. Thus, it suffices to

[4] The proposed OMG Trader Service is a step in this direction in OMG, but that would not solve the other limitations of the OMG object abstraction. In any event, the Trader also does not define a vocabulary.

determine only enough candidates that the Survivability Service has choices to allow it to select a "good" service instance to bind.

Once a set of potential service instances has been computed, one of them must be selected to be actually bound to the client. There are many criteria that can be used to make this selection (after all, any of them are semantically acceptable given the binding specification). For the purpose of constructing a survivable system, some of the considerations of the Survivability Service are to choose a service instance that can be instantiated in a robust way, uses minimal resources, and that can be easily reconfigured should it fail to meet the service binding requirement. This is a potentially very sophisticated tradeoff and is discussed further in *Evolution Support Toolset for Object Services Architectures*. Regardless of the policy followed, information contained in the models described in section 4.2 is required. This binding is where *semantic renegotiation* is introduced as a survivability mechanism.

Once a service instance to bind is selected, it must be *instantiated* before it can interact with a client. This is called a *service instance instantiation*. An important point is that unlike other OSA object abstractions, a service instance may have different instantiations at different times, and these instantiations may have very little physically in common. In particular, they may be constructed from different code bases, run on different hosts or O/S platforms, may be replicated with different multiplicity or with different replica coordination policies[5]. In order to be a candidate for binding, a service instance must have at least one possible instantiation.

A service instantiation is where the actual work of the service instance gets done. The service instantiation is responsible for implementing the methods defined by the service's interface type and for housekeeping to keep the service instantiation available. While there may be many ways to do this, we use the following pattern because it is simple to define and implement, corresponds to existing work in high availability services (see *Survivability in Object Services Architectures* for a discussion of this topic), and maps well to CORBA.

A *service instance instantiation* consists of a *service instantiation coordinator* and N *implementation objects*. An implementation object is an object in the object abstraction of an underlying implementation ORB (e.g., CORBA), which also provides *basic process control* for survivability. Being an object, it has an interface type, OID, and abstract state. Being at the implementation ORB level, these concepts are interpreted with reference to that object abstraction. The service instance instantiation provides *high service availability* as a survivability mechanism. The determination by the Survivability Service of the coordination policy, and the number, placement, and implementation class of implementation object instances provides *availability management*.

---

[5] None of this can be done directly in the CORBA object abstraction. While an IDL (interface) type may be implemented by multiple implementation classes, once an object is instantiated, it will always use the same implementation class and run out of the same "server" on the same host.

Appendix A-2

Each implementation object executes in exactly one process[6] and is the basic unit of failure. The implementation objects in a service instantiation are all instances of the same interface type, which is the same as that presented by the service. In general, the implementation objects may be of different implementation classes. The implementation objects are not individually visible to any entities other than the service coordinator and the implementation ORB (which must instantiate the implementation object and deliver messages). A client of the service instance interacts with the combination of coordinator and implementation objects in the same way it would interact with a unitary service instantiation in an implementation OSA like CORBA; e.g., normal looking surrogates and binding calls, single messages.

A service instantiation coordinator implements one or more coordination policies, each of which achieves a different objective with respect to making services robust. Some of these policies are voting (various schemes), hot backup, master-slave(s), and cold (persistent) backup. Each technique has a particular situation (defined by failure probabilities, likelihood of partition, relative frequency of reads to writes, etc.) in which it works better than others. Replica coordination technology is well known in the realm of *high availability services.*

The service instantiation coordinator does two things:

- it implements *view change management*: i.e., it keeps track of the current set of instantiated implementation objects, the role each is playing at any given time (e.g., master/slave or peers), and maintains the right number of active implementation objects by instantiating new implementation objects as needed, and
- it implements *replica coordination*; i.e., it receives messages from the client (directly or by trapping messages sent to one of the implementation object instances), routes messages to some subset of the implementation objects based on the chosen coordination technique, collects responses from the implementation object instances, reduces them to a single response, and returns that to the client. There may be more than one round of communication within the service instance instantiation depending on the coordination policy and responses (or lack thereof) from the implementation objects.

Because the implementation objects are distinct and may be of different implementation classes, any external services that they use need not be the same or even of the same type. In general, using different external services will make the instantiation more robust.

The coordinator can be independent of the type of the implementation objects it manages[7]. This is one of the key aspects of this pattern, since it allows services of any type to be made more robust by simply developing a small number of coordinator types.

---

[6] At least from the point of view of the ORB. If the object internally is implemented in multiple processes, this is invisible at our abstraction level.

[7] The coordinator need not know anything at all about the coordinated objects in order to coordinate them. However, it is desirable for the coordinator to know which messages cause reads and which cause writes, since this can be used to substantially reduce message traffic within a replica group. The meanings of the messages need not be known to the coordinator and are not interpreted. Type-specific coordinators can be

Appendix A-2

A coordinator is often implemented in a distributed fashion, with a piece of the coordinator residing with each implementation instance instantiation. This eliminates a single point of failure and allows messages from clients to be addressed to a coordinator residing with a nearby implementation object. Since, depending on the protocol, not all messages need be seen by all implementation instances, this reduces message traffic within the service instance instantiation. The coordinator could be a separate object residing with the instantiation, or it could be compiled into the implementation code using multiple inheritance or similar technique.

The pattern of a service instantiation coordinator and N implementation objects used to instantiate a service instance isolates the implementation of a service instance from any constraints imposed by the underlying implementation OSA. In particular, the restriction that an implementation level object is of fixed implementation class determined when the object is created is made irrelevant because there is no expectation that a service instance will always be instantiated using the same collection of implementation objects. All that is required to change implementation objects is a way to instantiate new implementation objects so that the abstract state of the service instance (as embodied in the concrete state of some valid implementation object or persistent state) is able to be transferred to the new implementation object. There are a variety of ways this can be done; they are discussed in *Evolution Model for Object Services Architectures.*

The pattern also isolates service instances from the choice of coordination policy. This is important since the circumstances that caused a particular policy to be chosen may no longer apply, in which case a different policy must be selected. For example, a policy that is very efficient when networks do not partition may become useless when network partitioning becomes common.

As an aside, note that this same pattern can be applied to coordination policies for goals other than high availability. For example, a service's data set could be partitioned among several implementation objects, each managing a portion of the data space. The coordinator would then transparently route requests to the appropriate implementation object. The coordinator would need to have a predicate informing it how to route requests, but would need no other information about the type.

4.2 Models

The following models, described below, are maintained.

- Resource Model
- Attack/Failure Model
- Threat Model
- Situation Model

For each, the major kinds of information contained in the model are described. Section 7 provides specifics of the models (i.e., E-R diagrams or class definition) as currently used

---

developed that take advantage of the semantics of the interface type to gain added efficiency, but this is not necessary, since the generic coordinators do quite well.

by the Survivability Service. That section will change and be expanded as we gain more implementation experience and learn which details are most useful.

It is important that the level of detail in the various models match to the sophistication of their use by the Survivability Service. For example, detailed failure models may be unnecessary if the failure detectors employed by the Survivability service are not themselves fine grained. This will probably result in similar levels of detail across the models at any given time.

4.2.1 Resource Model

The *Resource Model* describes the properties (interfaces and capabilities) of the various resources that are available for use or are already assigned to tasks. These are used to determine feasible places to instantiate a service instance. The following kinds of resources are of interest:

- Service Types
- Service Instances
- Service Instantiations
- Coordinator Patterns and Implementations
- Implementation Classes
- Implementation Object Instances
- Implementation Object Instantiations
- Interface and Implementation Repositories
- Hardware

4.2.1.1 Services Types

A language-neutral interface specification for interface types is specified in a type specification language that generally does not support implementation (i.e., has no executable code). In OMG CORBA, this language is IDL. Interface specifications are managed by an Interface Repository. Language-specific interface code is generated from these language-neutral interface specifications. We are not concerned with this aspect of ORBs, and plan on making no changes.

4.2.1.2 Service Instances

A record of all instances of each service type and their advertised properties is kept. A (very) tentative list of properties that can be specified is:

- *location* of either the service instantiation or physical devices (e.g., sensors or display devices) controlled by the service. It may be easier to specify that a service must be instantiated on some particular supercomputer than to specify detailed resource requirements that can only resolve to one host. In the case of controlled devices, actions in the physical world are not portable to the extent of operations in cyberspace.

- *abstract state* of the instance, especially when data partitioning of a large data set can be well defined (e.g., map services could be differentiated by the region of the world for which they manage maps). Service instances with the same interface and implementation that respectively manage maps of France and Spain are semantically different, even though the data may be in the same format and at the same resolution.
- *quality of service* that can (potentially) be provided by the instance. This should be interpreted in the widest sense of QoS, which includes concepts such as *resolution*, *currency*, and *accuracy*.

It seems desirable that the advertised characteristics either partition a space, form a hierarchy over a space, or create a DAG. Location and abstract state lend themselves to partitioning (e.g., maps of France or Germany) and hierarchy (e.g., maps of Europe, France, Paris), while QoS tends toward DAGs (e.g., 10 second delivery is better than 30 second delivery, but there is no obvious hierarchy of "goodness" over 10 second delivery with 3% errors or 30 second delivery with 1% errors).

If the properties that can be used for advertisements can be categorized like that, it should be possible to determine feasible services for initial bindings, equivalent services if rebinding is required, and approximately equivalent services if some requirement can be relaxed by the client. This is to be a major portion of our study in the next year of this effort.

4.2.1.3 Service Instantiations

The collection of currently instantiated service instances must be known, both to allow them to be reached and to ensure that the instantiation is unique.

4.2.1.4 Coordinator Patterns and Implementations

A description of the coordinator policies that are available for use and where to find their implementations. What each policy is good for needs to be described. This information will be used when instantiating a service instance in a given resource and threat environment. It is not clear what information is needed here, but it must be sufficient for the Survivability Service to avoid using a coordination policy that is unsuitable. Generally, such policies have been developed to address particular combinations of events such as "high read to write ratios in highly partitionable networks with infrequent server loss". An issue is how to state this kind of information in a way that can be matched with a given circumstance.

4.2.1.5 Implementation Classes

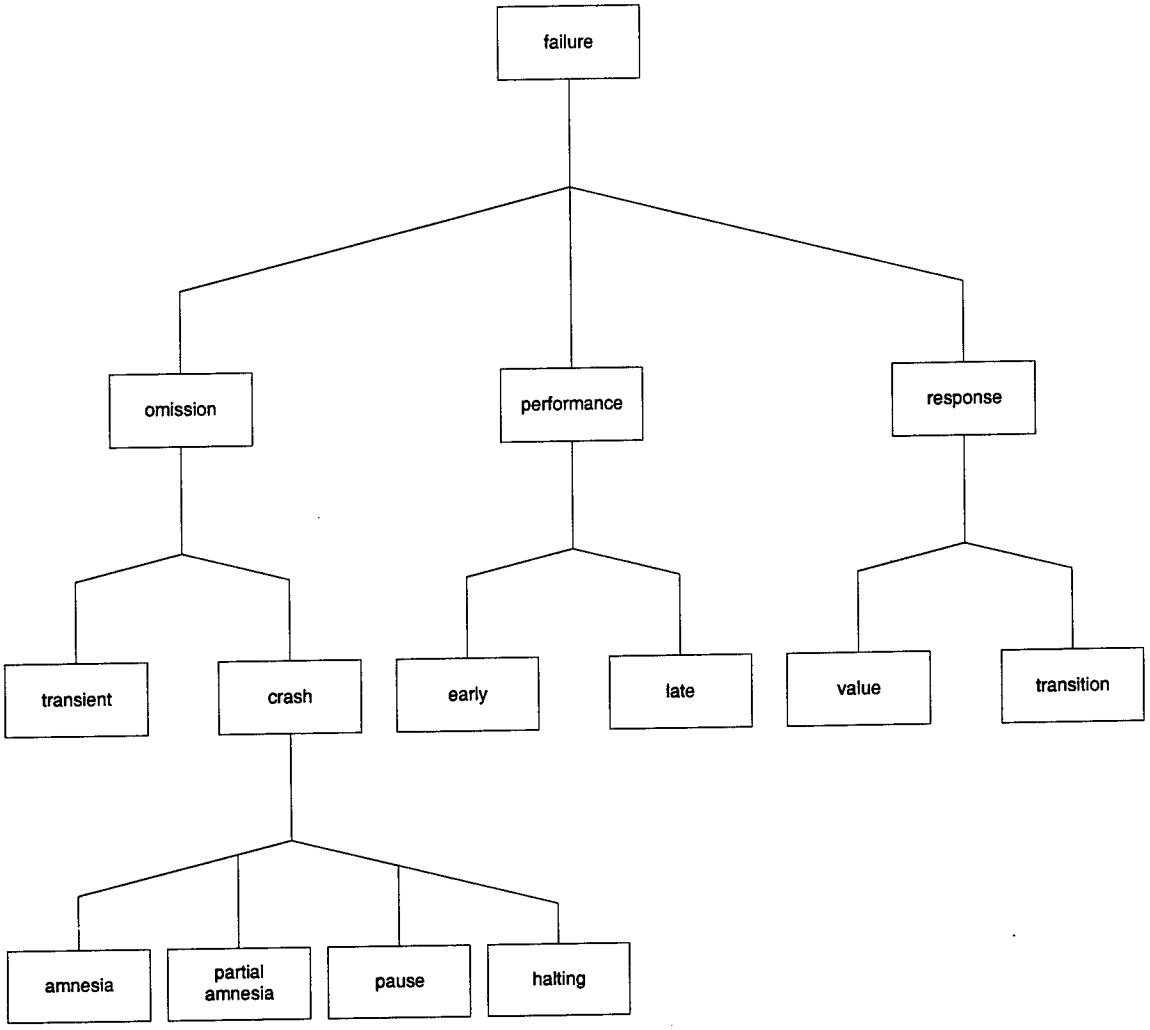
Implementation objects are instantiated as instances of some implementation class in the implementation level OSA. At that level, a given object has a fixed class that is determined when it is instantiated. Code for this class must be able to be loaded by the implementation ORB, which means it must be able to be located. Management of such information is the responsibility of the implementation OSA and is not discussed further here.

Appendix A-2

Here, we consider only additional information about the implementation classes that is relevant to the survivable object abstraction. This includes:

- *repositories* storing (or managing) the implementation class. This is used both to find the executables and to help determine whether a compromise of the repository could have compromised a particular copy of the implementation. For example, the Survivability Service might determine that code from certain sources has been implicated in an unusually large number of service failures and decide to use alternative sources if they are available, or perhaps modify the Threat Model to indicate that this code has a higher probability of failure than previously thought. This could then be reflected in an increase in the number and variety of replicas required to achieve a given availability.
- *environmental and resource requirements* of the implementation class. This includes such information as platform type, O/S, and other code to be linked. It can also include the quantity of resources required in order to provide various QoS that the implementation class chooses to advertise.
- *provenance* (or level of trust) of the implementation. This can be used in conjunction with the Threat Model to determine the likelihood of failure or attack against a given executable.
- *failure modes* of the implementation. Hierarchical masking and replication-based availability techniques work only when the probable failure modes of a given service are know. Each implementation is required to declare its failure modes. This will probably look something like a declaration of the exceptions a service can signal, although it will subsume that list, since a service can fail in ways that preclude signaling. Such declarations are expressible in IDL (i.e., there is a pre-defined OMG or Active-X accepted syntax for exceptions). The following is an example of a failure mode lattice proposed by Flaviu Cristian. Others are possible.

Appendix A-2



If implementation classes are designed so that they fail only in a small, predictable set of modes, implementation objects based upon them are called *fault tolerant*.

4.2.1.6 Implementation Object Instances

The implementation objects known to the implementation ORB, and their relationship to service instances are kept. Implementation object instances have meaning to the survivability object abstraction only as they implement service instances. Implementation objects can be constructed as needed, assuming that the correct concrete state can be part of the initialization. This state must come from either an existing implementation object or from stored persistent state.

4.2.1.7 Implementation Object Instantiations

It is useful to know which implementation objects are currently instantiated, since this reduces the cost of their use and may therefore change the choice of instance to use. This information is probably best kept by the service coordinators rather than in a global database.

Appendix A-2

4.2.1.8 Interface and Implementation Repositories

Code is stored in Implementation Repositories. The same implementation classes may be replicated across many repositories for performance and robustness. Information must be kept about the locations of the various repositories and about which executables (including version information) reside in which repositories. Since repositories are a source of vulnerability (they can be penetrated and their code corrupted or destroyed), it is necessary to keep information about the trustworthiness of repositories just as for implementation classes.

4.2.1.9 Hardware

This is a fairly straightforward description of hosts and networks. Their properties that are relevant to either scheduling tasks or classifying failures need to be maintained. The only real issue here is the level of detail that is useful. This needs to be determined based on the sophistication of the various users of the model.

4.2.2 Failure/Attack Model

The *Failure/Attack Model* describes potential failures of, or attacks against, system resources. Both failures and attacks cause resources to behave in ways other than desired. The model is used in the diagnosis of failures and helps to determine desirable configurations for survivability.

A failure or attack is described by a name, possible causal agents (who or what can cause the event), and the resources that can be affected. The terms failure and attack, although widely used, are somewhat misleading, since they imply unintentional and intentional acts. A better terminology would be to simply identify whether an event was intentional or unintentional, and if unintentional, has a benign external causative agent (e.g., operator error) or is caused by some internal failure (e.g., disk crash). This categorization is important, since intentional, malicious attacks are more likely to correlate than unintentional acts, and are less amenable to statistical analysis (e.g., we can envision "mean time between failure" measurements for hardware much more easily than "mean time between viral attacks". The two classes of events may be dealt with differently by the Survivability Service.

Some failures/attacks are service or implementation specific; these should have been identified in the Resource Model as the failure modes of the services or implementations. An exception raised in response to invalid inputs or crash-amnesia behavior are examples of these kinds of failures. Others attacks/failures are generic and can apply to any hardware or software. These are not part of the Resource Model, since they do not reflect designed behavior. Examples would be viruses, network errors outside the OSA model, and hardware failures.

One important issue with failures/attacks is the scope of resources affected by a single occurrence of the event or the class of resources potentially affected by recurrence of similar event. Threats to physical resources may be geographic in nature; for example, an attack on a particular site or sunspot activity. Ideally, an application should be

configured so that a single occurrence of a failure/attack of would not compromise all implementation objects in a service instance.

Threats to software resources are a bit more complex. There are three levels of software failure: instance, implementation, and service. Instance failure is what is typically thought of as failure. A particular service instance or implementation object instance fails for one reason or another. The failure is due to some cause external to the interface or implementation, for example loss of power, bad configuration of the environment in which the instance runs, or hardware failure or penetration that corrupts the code of the instance. Only that instance is affected, and it is unlikely that another instance can be attacked in the same way. Recovery involves restarting (after fixing the local environmental problems) or migrating the service or implementation object to somewhere that presumably does not have the environmental problems. Implementation failure and service failure are both higher level forms of failure. They indicate a fundamental problem with an implementation class or an interface type respectively. These sorts of failure would occur if a flaw were found in an implementation or interface that would allow an adversary to violate the implementation or service repeatedly. Implementation failure occurs if there is a bug in a program which could cause all instances of the implementation to fail. Multiple independent implementation attempts to solve this problem. The TCP open connection attack is an example of a service failure. Any instance of any implementation of TCP is subject to the attack. It cannot be prevented or recovered from within TCP. Implementation and service failures are not caused by the environment in which an instance of them executes. However, the environment can conceivably compensate for these kinds of failures (e.g., a security wrapper that prevents certain message traffic).

The Failure/Attack Model is necessary to support *failure detection and classification.*

4.2.3 Threat Model

A *threat* is a potential event, the *realization* of which could cause the loss or degradation of some set of system resources. If uncompensated, this could cause the system to malfunction. The realization of a threat is either an attack or a failure as previously defined in the Attack/Failure Model.

The *Threat Model* describes the threats to which a system is subject. This is basically the assignment of a likelihood or probability to an attack or failure. This can be described as a probability, a mean time between occurrences, or a fuzzy evaluation. All three should be supported, since depending on the threat, different amounts of information are known about its likelihood. Some threats such as hardware failure are quite predictable in the large, while others, such as the introduction of a virus, are very subjective. This description may be fairly simple or fairly complex. For instance, the Threat Model could identify not only the probability of a threat being realized, but the probability of its occurrence affecting particular resources.

The Threat Model is used to:

- help determine implementation object multiplicity for a given service instance instantiation. The number of implementation objects needed to ensure a desired level of service availability depends on how hostile the environment is.
- help determine implementation object placement. The objects should be placed to avoid threats if possible. Joint probability of failure based on threats is important here, since while the likelihood of a fire may be small, it makes little sense to place replicas where they can all be affected by the same fire.
- help determine how to evolve a system. The cause of a detected failure is an important consideration when determining a reconfiguration strategy. It is of course desirable to reconfigure in a way that avoids the same attack if that attack is considered likely by the Threat Model. It is also desirable to avoid reconfiguring in a way that makes the system vulnerable to other attacks as predicted by the Threat Model.
- attack/failure classification. The Threat Model is used in the estimation of the cause of a detected failure. Symptoms of a detected failure are used in conjunction with the threat probabilities to estimate which threat was the most likely cause. If multiple attacks could cause the same symptoms, the Threat Model can be used to estimate which is the most likely or to predict the most conservative course of action. For example, a node failure may be most likely the result of a race condition, but if a virus could also have caused the failure, it might be better to use a different load image when restarting, even if this is more costly.

Threats are generally classified by whether they are *intentional* (e.g., viruses or bombing) or *unintentional* (e.g., operator error or bad data) or *naturally occurring* (e.g., fire or earthquake). Despite the fact that the same kind of information is kept for all threats, this classification is important for several reasons. Naturally occurring attacks and failures are often well understood; insurance companies and the like have studied these for years and good statistics for them frequently exist. Furthermore, the severity of such threats is usually inversely proportional to their frequency. Unintentional attacks and failures are also fairly predictable, and it is unlikely that the success of an attack will cause others of the same nature. The opposite is true of intentional threats; success breeds success, and increases the likelihood of subsequent attacks of the same kind.

Many threat models assume that threats are independent. We believe this is an unrealistic simplification, since geographically oriented threats tend to occur in groups (e.g., power outages, storm damage), as do intentional threats as noted above. Thus, to some extent, either the Threat Model or the use of it should be stateful, although we will probably not do this in the near future.

The Threat Model will be evolved as the perception of threats changes. This may occur because of a change in the situation as defined by the Situation Model or by observation of attacks and failures that differs from what is predicted by the Threat Model. For example, the likelihood of physical loss of resources increases when at war or when a unit is in transit. Alternatively, direct observation may lead to a change in the perceived vulnerability to of attack, even though the underlying situation does not change. For example, if several identical service instances fail in a short period of time, it may be wise

to assume that the probability of threats against all instances of that type (such as programming errors or virus penetration) is greater than previously thought.

4.2.4 Situation Model

The *Situation Model* describes the real-world objectives to be met by the applications/services in various situations. A situation might be unloading a ship, launching a missile, or engaging in ground combat. For each situation, the Situation Model records how critical a particular application or service activity is. This provides objective functions that are used to rate the relative desirability of legitimate configurations. A change in situation may cause a change in the relative values of applications or services. For example, applications for managing the unloading of a container and scheduling where the material gets stored become relatively less important compared to an application for routing command messages to various consoles as conflict (or its potential) intensifies.

Another use of the Situation Model is that it is used to vary the Threat Model in response to changing situations. Depending on the situation, different threats may become more or less likely.

It is our understanding that something akin to a Situation Model is already created and maintained by the military for scheduling applications. There are several operational modes, including "normal", and several kinds of "alert" modes. For each situation, applications are assigned priorities that tell whether they must run, cannot run, or run if resources allow. The decision on which mode to be in is made by an officer. The mode to application matching is done well in advance and is a normal part of operations. Applications are aborted manually by an operator according to the policy. There are equivalent human actions as modes change, such as increased physical security, different stations to man, canceling leaves, etc.

5. OSA Implementation Architecture

The OSA Survivability Model must be mappable to an OSA Implementation Architecture. This means that every application architecture (and corresponding instantiation) that is constructed using the survivability model must be able to be instantiated in an OSA Implementation Architecture such as CORBA. This necessitates a discussion of the key constructs at the implementation level architecture. Our goal is to be able to map the Survivability Model to the three major OSA Implementation Architectures: CORBA, Active-X, and Java. At this point, we have considered only CORBA in any depth.

5.1 CORBA

Details of the CORBA object abstraction are well known. It is our intent to summarize the key features in this section when time permits.

5.2 Active-X

TBD

Appendix A-2

5.3 Java

TBD

6. Mapping the OSA Survivability Model to OSA Implementations

6.1 CORBA

We have determined approximately how this mapping will be done. At present, we have not actually implemented our design. It is our intent to document our current design as time permits.

6.2 Active-X

TDB

6.3 Java

TBD

7. Current OSA Survivability Model Specifications

7.1 Resource Model

Obviously for each kind of resource, there is an issue of how much and what kind of detail will be modeled. In general, more detail allows better fidelity to the real world and possibly better decisions at the cost of additional model complexity. Additional detail is useful only if it allows better decisions to be made. As a result, we will model each kind of resource to only as many levels of abstraction as can be used by the Survivability Service at any given point in its development.

Initially we will model all OSA services in the dependency graph of each simulated application. The only kinds of physical resources we will model are computers, network connections, and code images; there will be no composite physical resources considered.

The attributes to be modeled are driven by the attributes that can be used by the various Client and Resource Brokers when determining if a given level of QoS can be provided and how much to pay/charge for achieving that QoS. Since the initial version of the decision process will be very abstract, we will model each resource as follows.

Global Collections:

```
ComputerResources          : {Computer}

NetworkResources           : {Network}

Repositories               : {Repository}
```

Appendix A-2

```
Executables                 : {Executable}

ServiceTypes                : {ServiceType}

ServicesInstances           : {ServiceInstance}

ServiceInstantiations       : {ServiceInstantiation}

Servers                     : {Server}

ServerGroups                : {ServerGroup}
```

Simple Types:

```
PlatformTypes               = enum(Solaris, Linux, WinNT, Win95, Mac)

ReplicationPolicies         = enum(voting, primary/n-backup, ...)

ComputerQoSTags             = enum(memory, processorCycles, disk)

NetworkQoSTags              = enum(bandwidth, latency, errorRate)

ServiceQoSTags              = enum( TBD )

Location                    = (x, y : int)
```

Instances:

```
Computer (
      computerID            : OID,
      name                  : string,
      platformType          : PlatformTypes,
      location              : Location,
      memory                : int,   \\ in Mbytes
      processorCycles       : int,   \\ in MIPS
      disk                  : int,   \\ in Mbytes
      availMemory           : int,
      availCycles           : int,
      availDisk             : int
      )
```

Appendix A-2

```
Network (
      networkID            : OID,
      attachedComputers    : {Computer},
      bridges              : {Network},  \\ connected networks
      maxBandwidth         : int,
      availBandwidth       : int,
      latency              : int,
      errorRate            : int
      )


Repository (
      repositoryID         : OID,
      location             : Computer
      contains             : {Executable}
      )


Executable (
      executableID               : OID,
      className            : string,  \\ name of implementation class
      implements           : ServiceType,
      runsOn               : PlatformType,
      homeRepository       : Repository,  \\ where the code lives
      couldProvide         : {ExecutableQoSAssertion}
      )


ServiceType (
      typeID               : OID,
      typeName             : string,
      serviceInstances     : {ServiceInstance},
      implementedBy        : {Executable}
      )


ServiceInstance (
      serviceID            : OID,
      serviceType          : ServiceType,
      needsServices        : {QoSServiceSpec}, \\ services logically needed
      instantiatedBy       : ServiceInstantiation
      )


ServiceInstantiation (     {Abstract Class}
      instantiationID      : OID,
      instantiates         : ServiceInstance
      doesProvide          : InstantiationQoSAssertion
      )


Server : ServiceInstantiation (
      code                 : Executable,
      site                 : Computer
      )


ServerGroup : ServiceInstantiation (
      policy               : ReplcationPolicies,
      members              : {Server}
      )
```

Appendix A-2

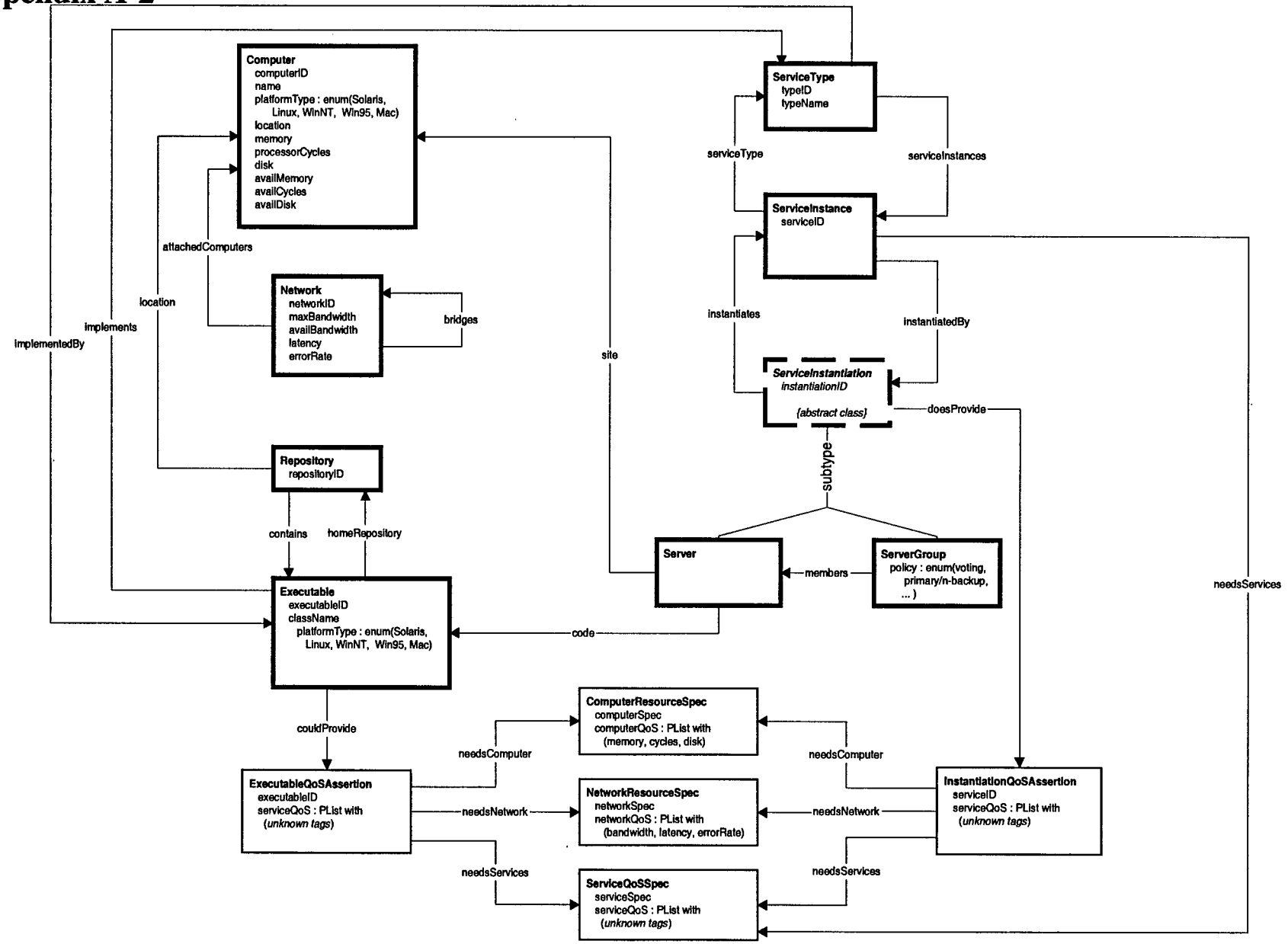
```
ExecutableQoSAssertion (
      executableID          : OID,
      serviceQoS            : P-list with unknown tags,
      needsComputer         : ComputerResourceSpec, \\ physical
      needsNetwork          : NetworkResourceSpec,  \\ physical
      needsServices         : {ServiceQoSSpec} \\ needed by implementation
      )
```

Note: this implies that each service instantiation reserves network resources to deviler its OWN results at the required QoS. No consideration is given to reserving network resources to deliver arguments to any services it may call. This is a complication we will avoid for now.

```
InstantiationQoSAssertion (
      serviceID             : OID,
      serviceQoS            : P-List with unknown tags,
      needsComputer         : ComputerResourceSpec,
      needsNetwork          : NetworkResourceSpec,
      needsServices         : {ServiceQoSSpec} \\ regardless why needed
      )


ComputerResourceSpec (
      computerSpec          : a predicate that resolves to a machine or
                              class of machines, e.g., "Machine4" or "WinNT"
      computerQoS           : P-List with tags in (memory, cycles, disk)
      )


NetworkResourceSpec (
      networkSpec           : a predicate that resolves to a network or
                              collection of networks, e.g., "Network3" or
                              "networks connecting Computer1 and Computer4"
                              or "Network with errorRate below 1/sec"
      networkQoSSpec        : P-List with tags in (bandwidth, latency,
                              errorRate)
      )


ServiceQoSSpec (
      serviceSpec           : a predicate that resolves to a service
                              instance or collection of service instances,
                              e.g., "MapServer1", "any MapServer", or
                              "any MapServer with 1-minute maps of Texas
                              from surveys no more than 10 years old"
      serviceQoS            : P-List with unknown tags,
      )
```

An E-R diagram for this appears on the next page.

Appendix A-2


Computer
computerID
name
platformType : enum(Solaris, Linux, WinNT, Win95, Mac)
location
memory
processorCycles
disk
availMemory
availCycles
availDisk
ServiceType
typeID
typeName
serviceType
serviceInstances
ServiceInstance
serviceID
attachedComputers
Network
networkID
maxBandwidth
availBandwidth
latency
errorRate
bridges
location
implements
implementedBy
site
instantiates
instantiatedBy
ServiceInstantiation
instantiationID
{abstract class}
doesProvide
subtypes
Repository
repositoryID
contains
homeRepository
Server
members
ServerGroup
policy : enum(voting, primary/n-backup, ...)
needsServices
Executable
executableID
className
platformType : enum(Solaris, Linux, WinNT, Win95, Mac)
code
couldProvide
ComputerResourceSpec
computerSpec
computerQoS : PList with (memory, cycles, disk)
needsComputer
needsComputer
ExecutableQoSAssertion
executableID
serviceQoS : PList with (unknown tags)
needsNetwork
NetworkResourceSpec
networkSpec
networkQoS : PList with (bandwidth, latency, errorRate)
needsNetwork
InstantiationQoSAssertion
serviceID
serviceQoS : PList with (unknown tags)
needsServices
needsServices
ServiceQoSSpec
serviceSpec
serviceQoS : PList with (unknown tags)

Appendix A-2

7.2 Failure/Attack Model

We have so far not defined the Failure/Attack Model in any depth. It is important to note that the level of detail to be maintained for this model depends on the sophistication of its two clients: the Survivability Service's Failure Detectors and Classifiers, and mechanisms the Survivability Service uses to configure and reconfigure applications.

Our understanding of Failure Detectors is that they are likely to be able to detect only the following kinds of failures other than ones reported by the service itself using exceptions:

- loss of site
- loss of service (other services at site still running)
- loss of access to site
- loss of access to service (can get to other services at site)
- degraded performance of site, service, or network (may not be able to differentiate)
- possible corruption of site or service (still providing some useable, but perhaps tainted service)

Configuration and reconfiguration actions taken by the Survivability Service are limited to choosing service instantiation coordination policies, assigning resources to implementation object instantiations, and choosing which implementation object (and classes) to use.

Both of these argue that the Failure/Attack Model should be at a fairly high level of granularity, since neither of its clients operates at fine grain.

7.3 Threat Model

TBD

7.4 Situation Model

TBD

Appendix A-3

Evolution Model for Object Services Architectures

David L. Wells, David E. Langworthy, Thomas J. Bannon,
Nancy E. Wells, Venu Vasudevan

Object Services and Consulting, Inc.

Dallas, TX

{wells, del, bannon, nwells, venu}@objs.com

Abstract

This report describes extensions to the Object Services Architecture model that make it possible to safely migrate a running application from one legitimate configuration into another legitimate configuration. Both semantically identical and semantically similar transformations are possible under this model, which allows applications to continue to survive in degraded mode when system resources become unavailable due to attack or failure. Legitimate transformations are determined based on the original application service binding specifications as described in the *Composition Model for OSAs* and mapping rules that define various possible transformations. From within the set of legal evolution possibilities, a number of system and threat models are used to determine a "good" transformation based on a malleable combination of predicted safety, best performance, and lowest cost.

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

Appendix A-3

Table of Contents

Appendix A-3

1. Introduction

An OSA-based application consists of a collection of object services interacting across an object bus. Not all possible configurations of OSA-based applications are equally robust, nor are all configurations equally able to be reconfigured. A companion paper, *Composition Model for Object Services Architectures* defines a subset of the possible OSA configurations that is more survivable and that can be reconfigured into other configurations in the feasible set. This paper defines legitimate evolutions of an OSA-based application from one of these desirable configurations to another. While it hints at the properties of "good" evolutions, detailed discussion of that topic is deferred to a separate paper, *Evolution Support Toolset for Object Services Architectures*, that presents an OSA Survivability Service that implements the models and chooses between legitimate configurations and evolution alternatives.

The paper is organized as follows. Section 2 summarizes the relationship between the OSA Composition and Evolution Models, and between our survivability object abstraction and the object abstraction presented by an implementation ORB such as. CORBA. Section 3 presents the various forms of evolution possible in the survivability object abstraction. Section 4 defines how these abstract evolutions are realized as concrete operations at the implementation ORB level. Section 5 describes how the models used evolve to reflect changing circumstances.

2. Overview of OSA Composition and Evolution

The OSA Composition Model defines an object abstraction in which to create OSA-based applications that are robust and are survivable through various forms of evolution. Applications defined using the OSA Composition Model are reified using the object model of an Implementation ORB such as CORBA. Mappings from constructs in the OSA Composition Model to an Implementation ORB are defined by the OSA Composition Model.

For a given application specification, there are potentially many reifications of it at both the OSA Survivability level and the Implementation ORB level. Of course to execute the application, one of these must be the one to be instantiated. This is done by the OSA Survivability Service using the facilities of the Implementation ORB. Instantiation may take place incrementally as an application progresses, and an application may relinquish resources it no longer needs. Applications compete with each other for resources.

Sometimes, an instantiated application may be forced to involuntarily relinquish resources before it is ready to do so. This may be because the resources themselves fail, or because some other application's competing demands are judged to be more important. In this case, the application must either be terminated, giving up all its resources and no longer providing any service at all, or it must be evolved in some way to use resources that are still available. It may be that after evolution, the application still provides the same service, or it may provide a similar service with degraded functionality. If either can be done, the application is called *survivable*.

Appendix A-3

Necessary and sufficient conditions for an application to be evolved are that there be another instantiation of the application that uses currently available resources, and that there be a state-preserving transformation from the existing instantiation to the proposed instantiation. Determination of possible new instantiations is done using the OSA Composition Model in exactly the same way as for the original instantiation, only using a different set of available resources. However, the mere existence of a different possible instantiation does not guarantee that it is legitimate. This is because work already done by the application is encoded in changes to the abstract state of the various services comprising the instantiation. Simply moving to a new instantiation that would have been a legitimate initial instantiation loses that information and is thus not legitimate.

The object abstraction defined by the OSA Composition Model has been designed to provide a number of "joints" where applications can be disassembled and reconfigured. The OSA Evolution Model uses these joints as places to apply a number of transformations. For each transformation, the OSA Evolution Model defines preconditions that must be met in order for the transformation to be applicable. These transformations not only construct a new instantiation of the application (usually an incremental change from the previous instantiation), but define actions necessary to make the abstract state of the various services compatible. These actions are performed at the level of the Implementation ORB in response to decisions made by the Survivability Service.

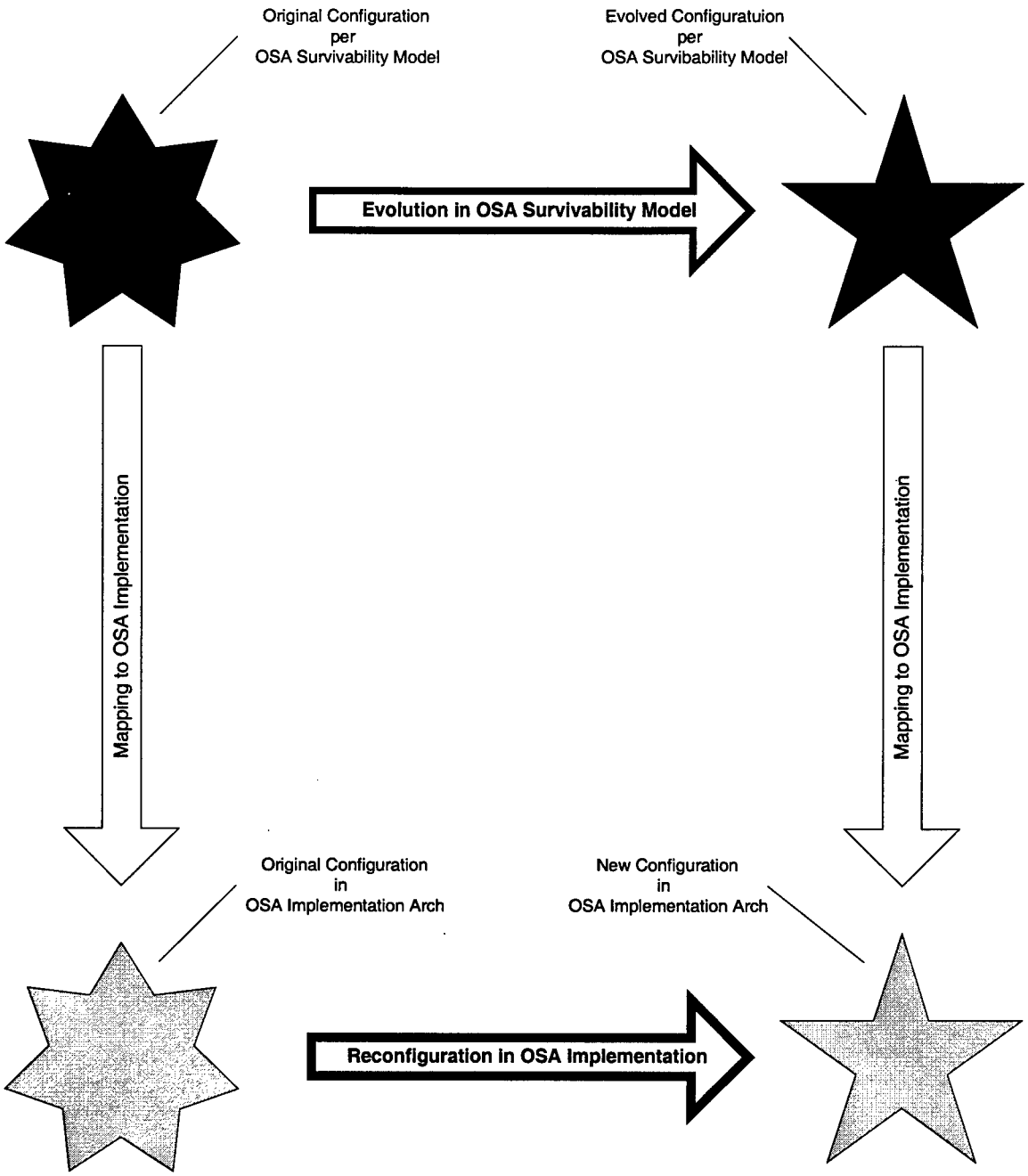
Just as the OSA Composition Model does not prescribe which of the legitimate configurations is "best" for some purpose, the OSA Evolution Model does not prescribe which evolution is "best" at some given time. The models have been designed to make the choice of a "good" configuration or evolution tractable, but this choice is properly outside the models[1], since different circumstances and objectives will yield different answers.

Each form of evolution has advantages and disadvantages. A given problem is potentially solved by many different kinds of evolution, and for a given type of evolution, there will typically be a set of legitimate possible outcomes. Determination of which alternative to use is the responsibility of the Survivability Service. In general, any form of evolution may be used by the Survivability Service to address any problem. In other words, there is no exact match between some kind of resource loss or objective change and a particular evolution action.

The following figure illustrates the relationship between the OSA Composition and Evolution Models and between the survivable and implementation object abstractions. The bold parts of the figure are those addressed by the OSA Evolution Model.

[1] It is made by the Survivability Service, described separately.

Appendix A-3

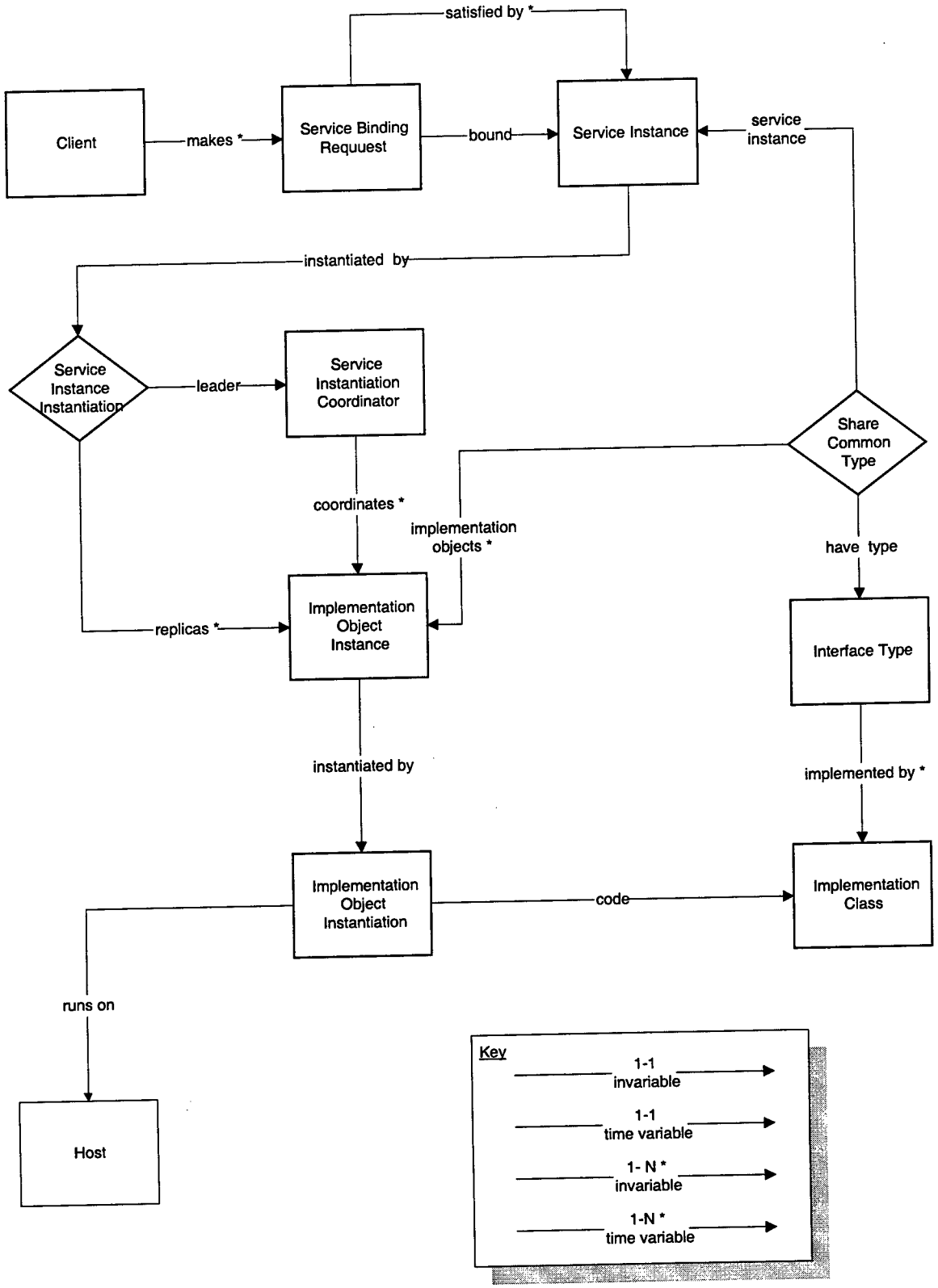


3. Evolution in the Survivability Object Abstraction

The OSA Composition Model defines an object abstraction with a number of places where evolution can take place. The figure below shows the object abstraction. Each time varying relationship (shown in the figure by a dashed line) is a place where a survivability transformation may take place. This section describes each of these "joints",

Appendix A-3

the kinds of transformations can take place each, and the preconditions that must be met before a particular transformation can take place.



Appendix A-3

A time-variable relationship can be changed after it has been made, whereas a time-invariable relationship cannot. In principle, any relationship could be time-variable, but this is often impractical, so some relationships end up being time-invariable. There are two reasons why a relationship might be made time-invariable: a need for efficiency in resolving the mapping defined by the relationship, and a need to ensure that state dependencies caused by a particular mapping are not violated if the mapping changes. If a mapping is time-invariable, it can effectively be cached and the perhaps complex process of determining the initial binding can be avoided. As an example, consider the difference in cost between a C++ and a CLOS method invocation. The need to preserve state dependencies is a bit more complex. When objects interact, they often affect each other's abstract state. This potentially mutual change of state is an encoding of the messages sent using a particular mapping for the relationship. If this mapping is changed, the states of all objects involved need to be reconciled. This includes objects in the original mapping and those in the new mapping. Doing this reconciliation is non-trivial, and may be impossible. Often, the places where this abstract state is concretely represented is known only to the object implementations, and cannot be reconciled from the outside. In any event, this reconciliation may be expensive and is not to be done lightly. By forcing a relationship to be time-invariable, the need to be able to perform this kind of reconciliation is eliminated.

The above discussion leads to two criteria that must be applied to changing time-variable mappings: it must be relatively efficient to determine the current state of the mapping and whether it has been changed, and there must be a way to reconcile mutual state for each of the kinds of binding changes that may occur. Note that the latter does not require the ability to do arbitrary reconciliation, since that is in general impossible. We sidestep this by only allowing transformations where we know how to reconcile state.

We now examine the time-invariable and time-variable bindings in turn. Time-invariable bindings are presented, even though they do not form part of the OSA Evolution Model, in order to justify why they are not also time-variable.

3.1. Time-Invariable Bindings

The time-invariable bindings are:

- Implementation Object Instance Implementation Class
- Implementation Object Instance Instantiation
- Service Instance Type

3.1.1. Implementation Object Instance Implementation Class

Implementation ORBs all require that an object instance (in this case an implementation object instance) always be of the implementation class under which it was instantiated. This simplifies code management at that level and presumably makes loading and initialization more efficient. Because this is buried deeply inside all Implementation

ORBs, we see no justification for attempting to change it. We achieve the important ability to vary implementation class in other ways.

3.1.2. Implementation Object Instance Instantiation

The concept of an implementation object instance instantiation is a bit slippery because it is not precisely defined at the level of the Implementation ORB. We take this to mean that the mechanism by which an instantiation is launched is fixed. In CORBA at least, this is embedded in a script registered with the Implementation Repository augmented by information automatically provided when an object instance is created. This process is too closed for us to want to try to modify it.

3.1.3. Service Instance Type

In principle, it would be desirable if a service instance could evolve its type or if a type itself could evolve. We have not addressed this issue, because of the difficulty of the problem and because the main thrust of survivability is the response to immediate problems engendered by loss of resources, and not on solutions that require programming effort.

3.2. Time-Variable Bindings

The time-variable bindings are:.

- Implementation Classes Implementing an Interface Type
- Host of an Implementation Object Instantiation
- Implementation Object Instances in a Service Instance Instantiation
- Coordinator of a Service Instance Instantiation
- Service Instance Instantiation
- Service Instance Bound to a Service Request
- Service Instances Satisfying a Binding Request
- Service Binding Requests by a Client

Transformations are presented from the "bottom up" in the model; i.e., transformations at lower levels are presented first. We do this is because lower level transformations are the most straightforward, generally cheapest to make, have the least impact on the application, and are therefore most frequently used.

All of these transformations allow the Survivability Service to do one of two basic things: to vary the way in which a service instance is instantiated, and to change the service instance to which a client is bound. They should be considered in light of those two higher level objectives.

Appendix A-3

3.2.1. Implementation Classes Implementing an Interface Type

New implementation classes can be added at will by simply registering them with the Resource Model and storing them appropriately. They can then be used when constructing new implementation objects. No reconciliation of state is required.

Implementation classes may be removed. This is likely to be done if either their code is lost or if the Survivability Service decides that it no longer trusts a particular implementation class. In the event that an implementation class is no longer available, none of the implementation object instances using that class can be instantiated. This may affect the ability to instantiate a service instance, depending on whether or not there are other implementation objects that can be used to instantiate it, and whether its state is stored somewhere else.

3.2.2. Host of an Implementation Object Instantiation

One way to vary a service instantiation is to change where its implementation object instances get instantiated. This does not change which implementation object instances are used, just where they are instantiated. This is an important distinction, because implementation objects are presumed to know how to create (or restore from persistent storage) their own concrete state, whereas if new implementation object instances were used, it would be the responsibility of the Survivability Service to initialize them property.

No reconciliation should be necessary if the location of the instantiation changes, although the implementation object may need to be more careful when restoring its state, since it cannot use default locations to find its state since it does not know where it will be instantiated.

Java certainly supports object portability. We believe that we can get a hook to allow the object to be instantiated on variable machines in CORBA, but have not yet tried it, so we don't know for certain.

3.2.3. Implementation Object Instances in a Service Instance Instantiation

In addition to changing where implementation objects are instantiated, it is possible to change the actual set of implementation objects used in a service instantiation. Two kinds of changes can be made: the number of implementation instances may be increased or decreased, or different (possibly newly created) implementation objects of different implementation classes can be used. Both kinds of transformations allow tailoring the service instantiation's performance and reliability. Having the implementation objects of different implementation classes increases robustness. Geographic distribution may increase or decrease reliability and performance depending on the coordination policy and the message traffic.

No reconciliation of client state is required, because this transformation is entirely transparent to the client. The service instance coordinator must be informed of the change in the implementation object instance set.

Appendix A-3

When an instance is added, the coordinator must bring the new implementation instance(s) up to a state consistent with the pre-existing implementation objects. There are two ways that this is usually done by coordinators, one (at least) of which *must* be supported: state transfer, and message replay[2]. State transfer is possible if at least one of the implementation instances that is up to date can externalize its state into a form that can be internalized by the new instance. This requires both a common external form and a pair of externalization/internalization functions in the implementation classes beyond what is required to implement the interface type. This approach is not universally applicable; in particular, if the concrete states of the two classes are very different, it might not be possible to define a mapping. An alternative is for the coordinator to initialize the new instance to some known state and replay to it the message traffic that has been sent to existing instances, thus allowing the new implementation instance to "catch up". This is only feasible if the message log is maintained far enough back that it meets the state to which the new instance can be initialized. Often this technique is augmented by checkpointing state occasionally. If both techniques are implemented, the coordinator can choose. Considerations in making this choice are the size of internal state and the length of message log that must be kept.

If an implementation object instance is to be removed, there must remain a sufficient set of implementation objects to meet the requirements of the service instance instantiation. This means sufficient abstract state to be functionally complete, and enough implementation objects instances to satisfy the coordination policy and provide sufficient QoS. While not an absolute requirement, it is preferable that the remaining set of instances support the addition of new instances at some future point. In particular, if an instance must be removed, it should not be the only one that knows how to externalize its state.

3.2.4. Coordinator of a Service Instance Instantiation

The coordinator of a service instance instantiation can be changed. There are several reasons why this may be necessary:

- the number of implementation objects may change sufficiently that either a new policy is needed (e.g., voting is no longer possible) or the existing policy becomes unacceptably inefficient,
- message traffic changes enough that a different coordination policy becomes more efficient (e.g., the ratio of reads to writes changes), or
- the threat situation changes enough that the existing coordination policy does not mask the threats (e.g., network partitions become so likely that certain kinds of voting are never able to achieve a quorum)

No reconciliation with the client is required, since this transformation is transparent to the client. A safe (and probably the only) way to ensure consistency among the

---

[2] State transfer is applicable only if the intent is to make the abstract state of the new instance match that of an existing instance. This would be the case for replication-based high availability services, but would not be for other coordination patterns such as data partitioning.

implementation instances during this transition is to temporarily block messages from the client until all internal messages to the implementation objects have been acknowledged and the implementation objects reach a mutually consistent state. This blockage may be unacceptable to the client, depending on its QoS requirements; if so, coordinator change is not possible at that time.

3.2.5. Service Instance Instantiation

Service instances need not be instantiated unless currently bound[3]. When reinstantiated, the new instantiation may be treated as either the same as the previous one (same OID at some level) or unrelated. The binding embedded into the client will reflect the current OID.

Since this is a new binding to a client, no reconciliation anywhere is required.

3.2.6. Service Instance Bound to a Service Request

A given service binding request may be satisfied by many service instances (see also section 3.2.7). Only one of these can be bound at any given time, but this binding can be changed under certain circumstances. In order to change such a binding, the client and new service instance must be synchronized to a mutually consistent state. In addition, the old service instance must be reconciled to the fact that it is no longer bound to the client. These present two different problems.

Reconciling the client and the new service instance means that the client must be able to handle responses from the new service instance. This does not necessarily mean that the client cannot tell the difference, although that is a desirable outcome. There are three cases, one of which must hold for a service instance to be rebound:

- the new service instance can be brought to a state compatible with the state of the old service instance,
- the client does not maintain state of the previous message traffic with the service, or
- the client can compensate for differences in mutual state.

We now consider generally applicable methods for achieving these conditions. Custom solutions are also possible, but are beyond our consideration.

There are three ways in which a new service instance state can be made consistent with the old service instance state:

- A stateless service achieves this condition vacuously.
- Message traffic to the old service instance can be maintained by the Survivability Service and replayed to the new service instance if some common staring point can be determined. Static initialization or initialization from a common persistent state would suffice. The message traffic must include messages from all clients, since this

[3] This allows reclamation of resources not actively required. On the other hand, a service instance may be instantiated all the time. This will typically be the case if initialization is time consuming, they are frequently bound for short periods, or their potential clients need very fast response.

is a way for interaction to occur. Care must be taken to avoid illegitimate transfer of information when doing this playback, so some security concerns hold.

- State can be transferred from the old service to the new service. It is not clear what the advantage would be of this, since it implies that the original service instance was still functioning. State transfer to a different implementation object instance (see 3.2.3) would accomplish the same without requiring service rebinding. This is, however, an option.

Stateless clients are quite possible for such things as signal processing or video display. Care must be taken to ensure that clients of that client do not receive the state.

Client compensation is necessarily client-specific.

Declarations of these conditions must be made part of the service specification (for both the client and service) in order for rebindings to be made and the appropriate technique to be applied.

Reconciling the old service to the loss of its client involves freeing resources and possibly reverting the service instance to a consistent state (it may have been left in an inconsistent state by client interactions that never completed). Reclamation of resources occurs naturally in some ORBs when an implementation object is not used for some length of time. In the event that the ORB does not perform this automatically or the resources in question are not allowed to be automatically released, the Survivability Service can manually request their release by the ORB.

Restoration of consistent state appears to be much like database transactions. From this, it would appear that the ability to wrap a stream of client messages in a transactional framework, with the Survivability Service having the ability to issue an "abort" if the binding is broken would be very useful. We have not considered the details.

3.2.7. Service Instances Satisfying a Binding Request

The set of service instances that advertise that they can satisfy a binding request varies continuously. This set need be computed only when a change in the service binding is contemplated. No reconciliation is required, since this just represents a set of possible choices from which to bind.

3.2.8. Service Binding Requests by a Client

For any given service needed by a client, the client may issue alternative service binding requests. These are independent binding requests for services that are perceptibly different to the client. Typically such requests are for the same kind of service at diminishing (or at least different) levels of QoS. A client will generally have preferences among these requests, accepting less desirable bindings only if this is all that is possible. As such, these binding request alternatives differ from a single service binding request that identifies many service instances; the latter identifies service instances that appear the same to the client, while the former identifies service instances that appear different to the client.

Such a change cannot be hidden from the client, and it is therefore the client's responsibility to compensate for the change. This will necessarily be client-specific, and poses the same set of issues as rebinding a service within a single service binding request (section 3.2.6). As in that transformation, it may be possible to synchronize client and service or it may be impossible. However, in all cases, the client must be able to handle the reduced QoS. This may be simple, as in a display routine, or may be more complex.

4. Reconfigurations in the Implementation OSA

Evolutions made within the survivability abstraction must be implemented by actions within and supported by the underlying Implementation ORB. In the Implementation ORB, far fewer facilities exist for evolution than at the higher level. This has two consequences. First, a single operation in the survivability abstraction will typically be implemented by several actions in the implementation ORB. Second, there will be no obvious relationship between the "before" and "after" configurations as seen from the implementation ORB's perspective; all such information is embodied in the survivability abstraction.

This section defines the actions that are taken in the implementation ORB to implement the various evolutions described in the previous section. There are three principal implementation ORBs, CORBA, Active-X, and Java. These are discussed separately.

So far, we have done an approximate mapping only to CORBA. Active-X is similar enough that the mapping should be similar. Java has different strengths and weaknesses, but we are convinced that a mapping to Java will also be possible.

4.1. CORBA

Details of this mapping will be provided in a subsequent report.

4.2. Active-X

TBD

4.3. Java

TBD

5. Model Evolution

In addition to evolution of an application, the models maintained in support of composition and evolution may also be evolved to reflect differing goals or understandings of system state. The kinds of evolution of these models that are likely are described here in brief. All changes to the models are made by (or through) the Survivability Service.

5.1. Resource Model

Changes in the resource pool are reflected in the Resource Model. Changes to the resource pool may force a reconfiguration or may change the reconfiguration choices

available to the Survivability Service. In general, resources may be added or removed, but the focus of survivability is generally on compensating for diminished resources.

5.1.1. Removing Resources

Resources may be removed from the resource pool because:

- they fail to perform properly, either by failing to respond, producing incorrect results, or by providing a lowered QoS, or
- they are deemed untrustworthy because attacks on, or failures of, similar resources lead to a belief that these resources are likely to fail or succumb to compromise in the future.

Resources that are reassigned away from an application or service because they are needed elsewhere, or that become inaccessible without themselves failing are not lost. Such changes are not reflected in changes to the Resource Model.

Failure of individual resources is detected by the Survivability Service's Failure Detectors and Classifiers. This may involve a considerable time lag and is not always accurate. For example, a resource may be assumed to be lost when it is only temporarily unreachable due to a misclassification. Because of these potential inaccuracies, there must be feedback loops within the Survivability Service to ensure that a planned action (based on the model) is actually able to be carried out. Resource loss can also be manually reported.

Resources can also be removed from use because they are suspected of being unreliable, even though they have not yet failed. This, of course, will not be a decision made by the Failure Detectors and Classifiers, since the resource in question has not actually failed. Instead, the Survivability Service examines the revised Threat Model, and decides whether it should remove a particular resource based on the estimated likelihood that it will fail or succumb to attack in the future. The kinds of resources that are likely to be removed in this way are those that have some common property. An example would be all implementation objects based on a (believed to be) compromised implementation class, or all instantiations on a particular machine. Resources may be speculatively removed from consideration when their use might cause damage in some way, for example by compromising data, causing a service using them to fail at some critical moment, or because they are so unreliable that the cost of using them exceeds the benefit gained.

Speculatively removed resources can be absolutely removed or can be marked in some way so that they are used only as a last resort. In the current implementation of the Survivability Service, it makes most sense to absolutely remove dangerous resources and to place a low value on the use of unstable resources and allow the Market to denigrate their use except in dire circumstances.

5.1.2. Adding Resources

Resources may be added at any time as well, although the general assumption for survivability is that this will be infrequent and will be limited to only certain kinds of resources. As such, the OSA Evolution Model does not provide support for activities

such as software development that are unlikely to occur when a system is under attack. However, the fruits of such development can be used, as long as they appear as a ready resource of a type already familiar to the Survivability Service.

Any physical resource can be added at any time. A very common occurrence is that a previously failed resource will become available again; e.g., a host fails and reboots, or communication with a mobile host is restored.

Implementation objects are routinely instantiated when they are touched. New implementation object instances can be created at any time by the Survivability Service.

Implementation classes, interface types, and coordinators are much less likely to be added, since they cannot be created automatically by the Survivability Service and generally require programming effort to create. Similarly, service instances are infrequently created, since this requires knowledge of why they are being created and what their initial abstract state should be, which is again a human activity normally outside the scope of survivability.

5.2. Failure/Attack Model

The Failure/Attack Model is changed very infrequently and only by human intervention to redefine failures or attacks or to define new ones.

5.3. Threat Model

The purpose of the Threat Model is predictive: to avoid configurations that are likely to become bad, and to aid in failure diagnosis. The Threat Model changes when one of two things happens:

- a new failure or attack is determined to exist, or
- the perceived likelihood of an existing threat changes.

The entry of new threats is discussed in *Composition Model for Object Services Architectures* and is not discussed further here.

The perceived likelihood of threats can change because either:

- the real-world situation changes, making particular threats more or less likely, or
- analysis of past attacks/failures indicate that the threat model was incorrect.

Changes to the Threat Model based on changes in the real world are embodied in the definition of the Situation Model. It is not clear whether the Situation Model should define threat likelihood explicitly for given situations, or provide a more abstract definition that is used to bias the computation of threat likelihoods.

Modifications to the Threat Model based on analysis of attacks/failures are more complex. They start with the detection of *symptoms* of problems detected by Failure Detectors. These symptoms are events that can be directly observed by a detector, such as a service that has not responded within a certain amount of time, an error rate that has reached some threshold, etc. From this, the Failure Classifier attempts to determine

which resource(s) have failed and in what *mode*[4]. At this point, a failed resource is identified, but in general, the cause of the failure is not known. Misclassification of failure mode or even of which resource failed are possible, and may be common. Classifiers may choose a broader failure mode than is actually the case (e.g., report node failure when the problem is actually the failure of some service at the node), or report an error in a resource when the actual error is upstream (e.g., report a host failed when the problem is in the network).

At this point, it is known (with potential for error) which resource is no longer adding value; this may cause the resource to be replaced. This is the point where the Resource Model is modified, but as yet, no changes occur to the Threat Model.

The underlying cause of resource failures are what we term attacks/failures[5]. The next step is to attempt to determine which attack/failure caused the resource to fail. This is generally a time consuming, off-line process that requires human involvement; e.g., through something like CERT. It may be very straightforward, however, such as a report of a bombing or the cutting of a wire with a backhoe[6]. Most difficult to determine are attacks/failures that can affect multiple resources, since the correlation of failures of individual resources into the underlying patterns is difficult and usually requires rather lengthy event logs, which of course also contain totally unrelated events.

Regardless of how the determination of an attack/failure is made, the next step is to decide if this should cause a change in the Threat Model. The Threat Model predicts how frequent a failure type should be. It is only when observed failures deviate from this by a significant amount that the Threat Model should change. We cannot be too precise about what this means at present because we are not sure how to describe threat likelihoods for various kinds of threats. For more discussion of this, see *Composition Model for Object Services Architectures*, section 4.2.2. As an example, likelihood of hardware failures is probably couched in terms of a mean time between failures provided by the manufacturer, while likelihood of a virus attack is a more subjective value. Modifications to MTBF may be made because of a hostile physical environment and can be based on direct observation or an informed estimate based on similar hardware in similar environments. Modifications to the likelihood of a virus attack are much more complicated. The initial estimate is probably based on such factors as software development rigor, sources from which downloads are accepted, and virus screening performed, without any knowledge of whether a virus attack on the particular software actually exists. Once a virus attack against the particular software has been detected, the likelihood estimate changes radically, since an attack can be made whenever desired by the possessor of the virus, assuming it is not screened out by download restrictions or virus screening.

---

[4] See *Composition Model for OSAs*, section 4.2.1 for a discussion of failure modes. Basically, it is the type of failure from a predefined set.

[5] We need a better terminology here, since the word "failure" is used for both the failure of a resource and its underlying cause.

[6] In principle, it would be desirable to be able to infer causes of failures by correlating resource failures with the set of attack/failures that could have caused them to determine patterns. This is outside of our scope, and in any event is likely to yield only suggestions to a human analyst.

Appendix A-3

One possible way of modifying the Threat Model is to increase the likelihood estimate by a small amount every time an attack/ failure is detected and reduce it periodically if nothing occurs. We don't know if an adaptive approach like this is reasonable, but it appears worth investigating if no better method is found.

5.4. Situation Model

The Situation Model is changed very infrequently and only by human intervention to redefine situations or to define new ones.

Appendix A-4

OSA Survivability Service

David L. Wells and David E. Langworthy
Object Services and Consulting, Inc.
Dallas, TX
{wells, del}@objs.com

Abstract

This report describes the architecture of an OSA Survivability Service that uses the *OSA Composition Model* to initially configure OSA-based applications and reconfigures them for survivability using the *OSA Evolution Model*. The Survivability Service uses a single set of system models and specifications for both purposes. The Survivability Service is compatible with existing work in failure detection and classification, fault tolerance, and highly available systems. Both the internal architecture of the Survivability Service and its connections to external services are described. Portions of the Survivability Service are being prototyped as part of this project.

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

Appendix A-4

Table of Contents

Appendix A-4

1. Introduction

Our goal in the design of the Survivability Architecture is to take system integrity to the next level. Fault tolerance enabled reliable data storage. High availability enabled reliable on-line processing. These technologies ensure islands of availability. Survivability enables reliable service delivery. The focus moves from hardening individual components to ensuring that every client has access to the services it requires. The Survivability Architecture seeks to provide continuity over discrete highly available systems though extensions to an object service architecture.

Consider planning a sortie for a regional conflict in which there are multiple coalition partners. Mission planning requires many resources, among them a map server. Assume the local map server becomes unavailable. The backup map server is located at a remote location and reachable only over slow communication lines. There is a coalition map server available with good performance characteristics, but while this service is signature compatible, its data is considered to be of lower quality and the labels are specified in a foreign language. Existing OSA implementations cannot switch an active connection and are limited to exact substitutes for a service. A survivable OSA needs to be able to switch compatible services in an established connection and substitute acceptable alternatives.

The ability to substitute services does not in itself elevate an OSA to a survivable OSA. Consider an information warfare attack which focused on NT machines. As the NT machines began to fail, essential processing must be moved over to UNIX machines. This in turn requires terminating or delaying non essential processing on those machines. This particular adaptation would not be difficult to implement in an ad hoc manner. However, there are many different threats, each with their own optimal response, and more than one threat may materialize at the same time. A survivable OSA must be able to dynamically adapt to the threats in its environment to reallocate essential processing to the most robust resources.

The remainder of the paper is organized as follows. Section 2 presents an overview of the Survivability Service, including the context in which it is intended to be used, its internal organization and major components, and external services it relies upon. Sections 3-5 describe the Survivability Service's major components (Enhanced Binding Service, Survivability Management Service, and Models) in more depth. Section 6 discusses Related Work. Two other reports produced by this project, *Composition Model for Object Services Architectures* and *Evolution Model for Object Services Architectures*, should be read before reading this paper. Many of the terms used in this paper are defined in those reports and are not redefined here.

The Survivability Service is still very much a work-in-progress. As such, some portions of the design are still underspecified and it is expected that the design will evolve as we gain implementation experience over the next year. Portions of the system have been prototyped. Even by the end of the project we will not have implemented our entire design due to resource limitations.

Appendix A-4

2. Survivability Service Overview

This section discusses the general context in which the Survivability Service is intended to operate, its internal organization, the kinds of survivability actions that are combined in our system, and how the Survivability Service uses a higher level object abstraction in which survivable configurations are specified and maintained.

2.1. Context and Operation

The Survivability Service creates and evolves "legitimate" survivable service and application configurations. Legitimate configurations are defined in a *survivable OSA object abstraction* defined in *Composition Model for Object Services Architectures* and *Evolution Model for Object Services Architectures*. In addition, the Survivability Service can provide full benefit only to applications specified in the survivable OSA abstraction in environments where resource, threat, and situation modeling has been done. Without these, the full range of survivability actions is not possible and the allocation and placement decisions cannot be made sufficiently accurate to assure non-brittle configurations.

The environment in which the Survivability service operates is shown in the figure below.

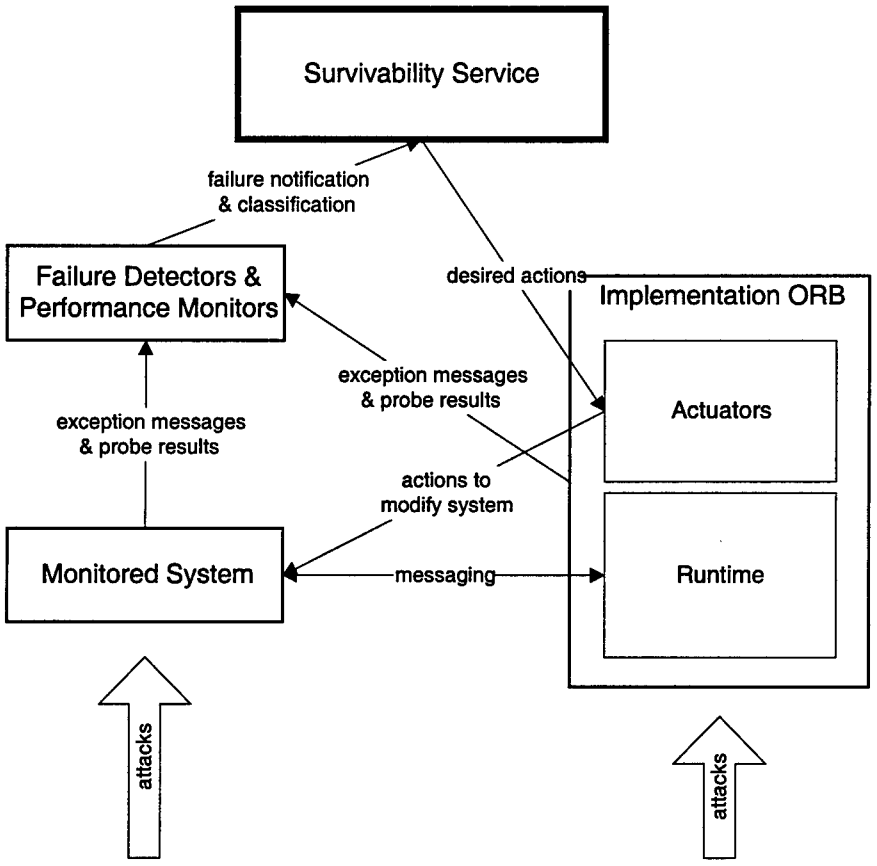


Figure: Context of the Survivability Service

Appendix A-4

The *monitored system* consists of a collection of objects interacting across a conventional *implementation ORB*. The objects in the monitored system are specified in a basic OSA abstraction, which does little or nothing to provide survivability. The objects may be organized into a number of independently developed (and hence non-cooperating) applications. As a consequence, objects and applications at the level of the monitored system cannot properly provide for their own survival because there need not be an overarching design for them and certainly no global coordination.

The *implementation ORB* provides the infrastructure on which the monitored system executes. This includes facilities for storing object interface definitions and implementations, locating and binding objects, activating and deactivating objects, and delivering messages and returning results. The object abstraction presented by the implementation ORB and the implementation ORB itself have severe limitations that make it difficult to build survivable services and applications directly at this level.

The monitored system is subject to *attacks* against its objects that may cause them to fail to perform as specified in a variety of ways, including crashes, degraded performance, denial of service and disconnection, or Byzantine failures. The attacks may be malicious or accidental. Objects may be attacked by attacking any of the resources they use, including their code, state (runtime or persistent), or physical resources (e.g., CPU cycles, disk, network access).

Failure detectors and *performance monitors* observe the monitored system to detect deviations from desired behavior. They do their best to identify that an object has failed, the mode of the object's failure (how it failed), which resources caused the object to fail, and the underlying cause of the failure (which attack caused the failure). Detection and categorization is by no means complete, precise, accurate, or timely. Failures may be undetected or may not be detected until some significant time after they occurred. The failure mode may be misclassified (hopefully too broadly but not incorrectly) and details of which resources failed may be unknown. Except when failures have obvious causes (e.g., the building containing a machine was bombed) it is unlikely that the underlying cause of the failure will be known until much later, if ever. This will be especially true in intentional software attacks, in which identification may require consultation with a CERT-like organization.

The *Survivability Service* maintains information about:

- the current state of the monitored system (as reported by the monitors and hence not necessarily accurate),
- configuration and survivability objectives,
- estimates of the threat of various kinds of attacks, and
- the kinds of reconfigurations that can be implemented using the capabilities of the implementation ORB under which the monitored system is running.

Using this information, the Survivability Service determines actions to take to move the existing configuration of the monitored system toward a more desirable configuration. The Survivability Service then issues commands to the implementation ORB in order to carry out those actions. If these actions fail, the Survivability Service will either receive a report from the implementation ORB or will rely on the failure monitors.

Appendix A-4

The internal organization of the Survivability Service is shown below.

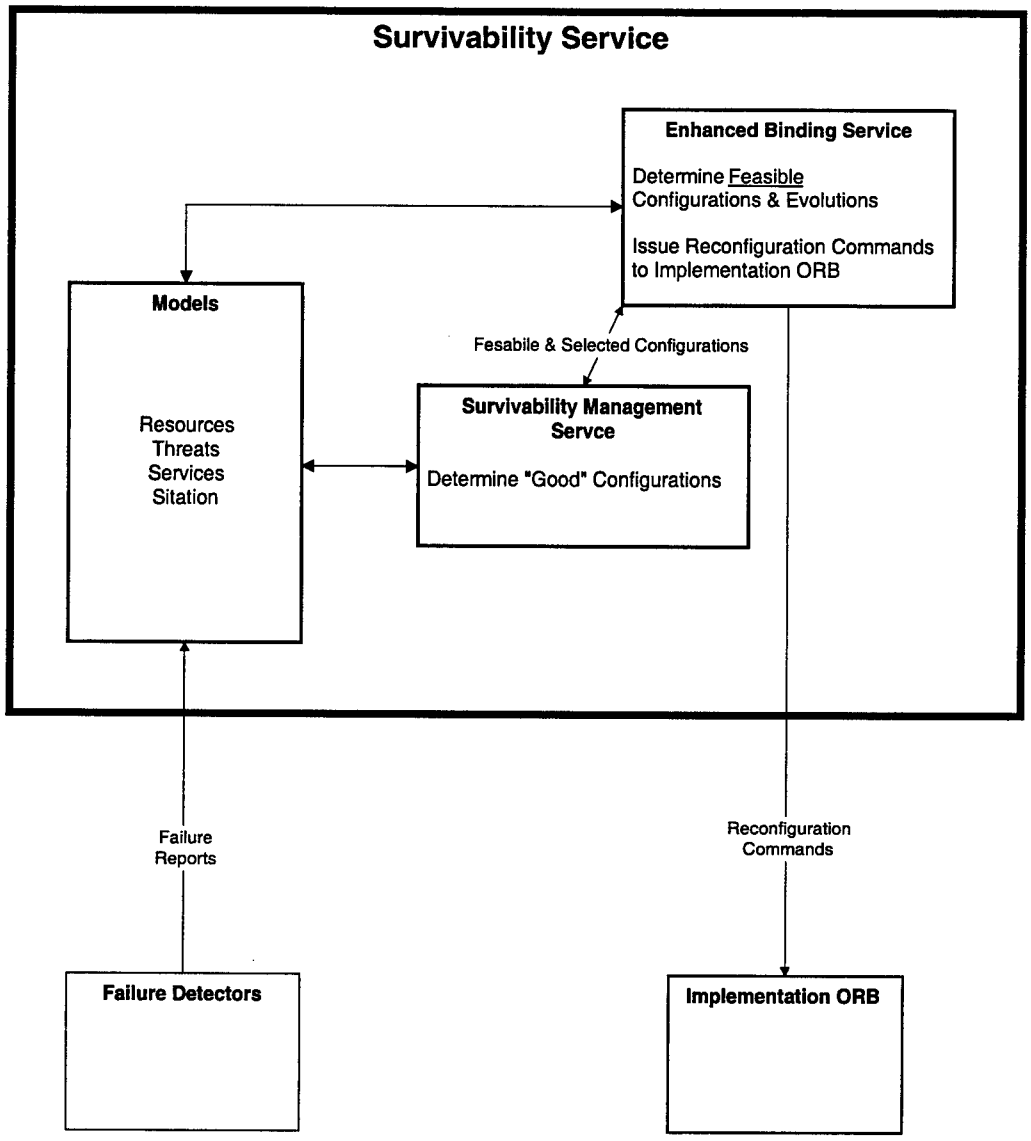


Figure: Survivability Service Internal Organization

The Enhanced Binding Service (EBS) opens up the standard OSA object abstraction to provide more alternatives for reorganizing clients and services and provides the mechanisms to perform the reorganizations. The Survivability Management Service (SMS) selects desirable configurations from among those enabled by the EBS and informs the EBS, which then uses the facilities of the implementation ORB to perform the actual reorganization. Models contain the information used by the EBS and SMS to determine feasible and desirable configurations and evolutions.

Appendix A-4

2.2. Unifying Approaches to Survivability

Survivability is achieved by combining many separate techniques, each of which is most appropriate in a different situation. Robust and efficient survivability requires the following capabilities, arranged roughly in a hierarchy:

Basic Process Control: The ability to start, stop and restart processes, to clean up after failed or aborted processes, and to restore processes to known states. Most of this is provided by *implementation ORBs*.

Fault Tolerant Services: These are services designed to (usually) fail in known "good" ways. Their failure modes become part of the *service specification*. This must be provided by the service developers.

Failure Detection & Classification: These mechanisms detect the symptoms of failures and attacks, and classify the events into likely failure categories. This can be done through probes, wrappers, or exception reports from well-behaved services. Classifying observed symptoms into error categories is at least partially based on the failure mode specifications of the fault tolerant services. *Failure detectors* are an external component relied upon by the Survivability Service. We are not working in this area, and will either obtain these mechanisms from elsewhere or assume an oracle for demonstration purposes.

High Service Availability: These are a collection of mechanisms to make individual service instances much more highly available than they would otherwise be. Techniques are either based on replication or hierarchical masking (i.e., error handling in the client). We concentrate on using replication-based policies since they do not rely on the semantics of the services and are therefore more widely applicable. Many replication-based policies exist (e.g., voting, hot backup, error correction) and some are integrated with ORBs (e.g., Electra, Eternity, and the late Orbix+Isis). These mechanisms must be efficient since they are invoked during normal (non-error) operation. At this level, it becomes possible to physically reconfigure an application by changing the way individual services are implemented. The logical organization remains fixed in that clients still interact with the same services after any reconfiguration. We assume that the *implementation ORB* will provide these capabilities, since they are showing up in research systems and will eventually migrate into products, either as part of the ORB or as attached CORBA services. We refer to ORBs with these capabilities as *fault tolerant ORBS*.

Availability Management: This layer manages the use of the High Service Availability mechanisms. It determines the appropriate fault tolerance mechanism to use for a given service based on service failure modes and perceived threats, and determines the resource pool needed to achieve desired availability. It can be less efficient than the lower layers since its use is infrequent or can be a background activity. This functionality is provided by the *Survivability Service*.

Service Renegotiation: At this level, it becomes possible to change the logical organization of an application by binding clients to alternate services if the desired service should become unavailable or degrades in performance. The rebinding can be to an equivalent, but distinct service (e.g., a different server having the same maps), or to a similar, but acceptable service (e.g., a different server with maps of the same area but at

lower resolution). Alternatively, the same service connection can be maintained but at a lower quality of service (e.g., more errors or slower). This is semantically more sophisticated than lower layers and requires specifications of client-service connections beyond those currently used in OSAs. In addition to allowing rebinding to service alternatives when services fail, service renegotiation can represent a fallback position if the costs of assuring service availability become unacceptably high. This functionality is provided by the *Survivability Service.*

The capabilities supplied by the implementation ORB and the various parts of the Survivability Service are summarized in the following table.

	Fault Tolerant Implementation ORB		**Survivability Service**	
	Basic ORB	**High Avail Services**	**Availability Management**	**Service Renegotiation**
Service Instantiation Type Choice	Constant		Static	Dynamic
Service Startup	Manual & On Open	On Failure		
Server Replicas per Service	1	N w/ Fixed Membership	N w/ Variable Membership	
Replication Policies	0	Fixed	Selectable	
Logical Connectivity	Fixed			Variable & Flexible
Model & Decision Sophistication	Path Completion		Resource Model	Semantic Models
Mode of Survivability Actions	None	Reactive	Proactive	
QoS	None		Fixed	Degradable

Figure: Locus of Survivability Actions

The Survivability Service provides both Availability Management and Service (Re)Negotiation. Both the Enhanced Binding Service and the Survivability Management Service components of the Survivability Service are required in the provision of each of these types of survivability actions. This is shown by the figure below.

Appendix A-4

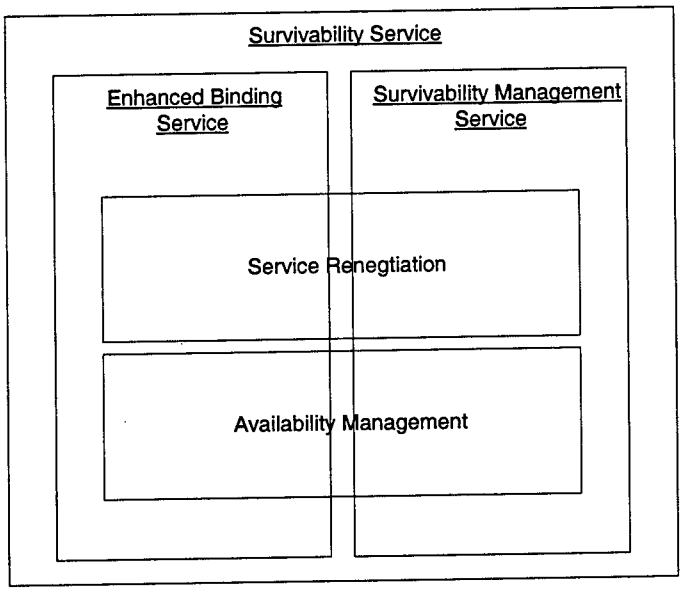


Figure: Mapping Levels of Survivability to the Survivability Service Components

To date, we have concentrated on the survivable object abstraction, the high level organization of the Survivability Service, and mechanisms for providing availability management. Our detailed architecture and prototyping have focused on the Survivability Management Service aspects of availability management (this corresponds to the lower right quadrant of the above figure). In the next year, we will devote most of our effort to service renegotiation, and will sketch how the EBS supports availability management. The latter appears straightforward.

3. Enhanced Binding Service

As noted previously, the *basic OSA abstraction* defined by implementation ORBs is insufficient for the specification and maintenance of survivable systems. Aside from a number of ambiguities, there simply is not enough flexibility in how systems are configured or reconfigured. In particular, reconfiguration of a running system in the event of failure receives no support from standard ORBs. Fault tolerant ORBs add some reconfiguration ability, but it is limited to the management of replicas of individual services. The required flexibility is added by the EBS, which:

- defines a *survivable OSA abstraction* in which a wide variety of survivable configurations can be defined and maintained,
- uses resource models and enhanced binding specifications (part of the survivable OSA abstraction) to determine feasible configurations and evolutions of existing configurations,
- provides the Survivability Management Service (SMS) component information about feasible configurations and evolutions and accepts back the SMS's determination of the desired new configuration, and
- issues commands to the implementation ORB to create the desired new configuration.

Appendix A-4

3.1. Survivable Object Abstraction

To overcome limitations of the standard OSA object abstraction, we have designed a *survivable OSA abstraction* (below) in which a wide variety of survivable configurations can be defined and maintained. The specified configurations are implemented using the basic OSA abstraction and capabilities provided by some underlying "implementation ORB" such as CORBA.

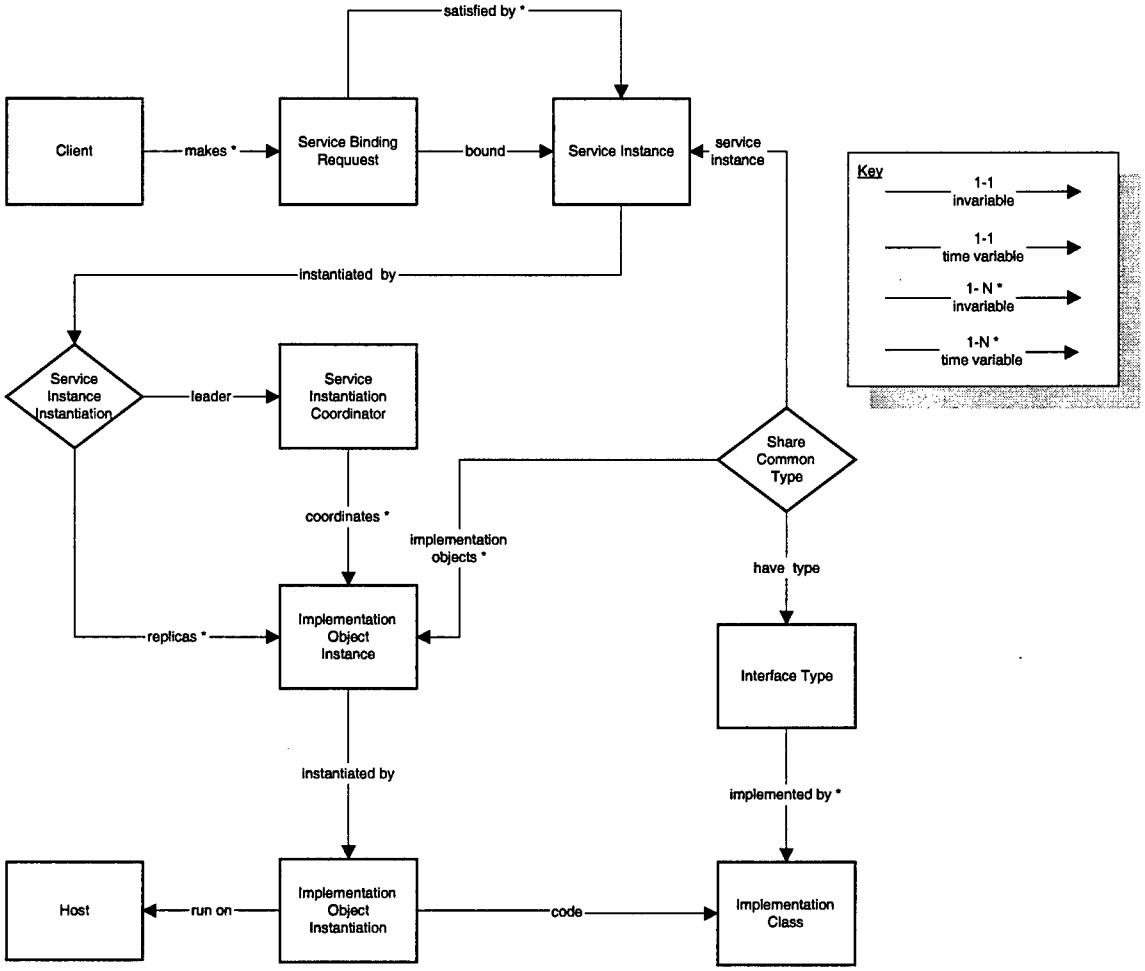


Figure: Survivable OSA Abstraction

The purpose of the survivable OSA abstraction is threefold:

- it clarifies some ambiguities in the basic OSA abstraction,
- it adds constructs to make it easier for programmers to specify survivable object configurations (and more difficult to specify non-survivable ones), and
- services and applications specified using it can be managed for survivability by a Survivability Service.

Appendix A-4

The survivable OSA abstraction defines several kinds of logical and physical entities, including abstractions such as services, instantiations of servers that implement the services, code images for the servers, and physical resources such as hosts and networks. A variety of relationships are defined between entities of these types. Most of the relationships can change and many have arbitrary cardinality. The Survivability Service manages these entities and relationships, creating, changing, and destroying them as necessary to create and maintain "good" configurations. The entities and relationships are defined in *Composition Model for Object Services Architectures*; changes to them are defined in *Evolution Model for Object Services Architectures.*

3.2. EBS Connections to Implementation ORBs

The EBS interacts with the implementation ORB at two levels: to implement availability management and to implement service renegotiation. The mechanisms at these two levels are not thoroughly designed yet, but are expected to take the following form:

Availability management requires starting and stopping CORBA-level services in appropriate locations with appropriate resources as dictated by the SMS. All relevant ORB hooks appear to be provided by CORBA Naming Service, Implementation Repository, and Loaders.

Clients must also be capable of being rebound to logically equivalent CORAB-level services. This does not require any synchronization, but does require the binding to be changed. This should be doable either by a level of indirection or by a modification to the client-side dispatch mechanism. In addition, the EBS must be able to inform the SMS which physical resources and CORBA-level services can be used to satisfy a particular request. This is embedded in the resource model, the details of which are not yet fixed. Note that at this level, service alternatives are not considered; it is assumed that the service instance to be bound was either stated explicitly by a client or was determined at the service renegotiation level of the Survivability Service. All relevant ORB hooks appear to be provided by CORBA Naming Service, Loaders, ORB interceptors, compiler-generated client stubs, and the open() binding call.

Service renegotiation allows alternate service instances to be specified and bound. Since the standard open() command in CORBA does not support this, we will have to extend that interface or provide an alternative. The OMG Trader interface is an obvious place to hang the required functionality, but the Trader specification mostly addresses "how" to utter such a request rather than "what" such a request should contain. The vocabulary and interpretation of the enhanced binding request needs to be determined. Some sort of "matching" capability is also required to determine exact and approximate matches. The better this matcher is, the more configuration choices are available.

Service renegotiation also requires the ability to change the logical service to which a client is bound. This will frequently require a synchronization of client and service state. There are a variety of ways this can be done, including message replay. A useful optimization can be made if meta data indicates that a connection is known to be stateless, in which case synchronization, and its associated overhead, can be avoided. Actually changing the binding should be able to use the same physical mechanisms as used to change bindings for availability management.

Appendix A-4

4. Survivability Management Service

The Survivability Management Service (SMS), maintains a balance between resources and demands as both dynamically change. The SMS guarantees that essential clients have the resources they need without any short-falls due to excess capacity for a non-essential purpose.

An essential property of the SMS is that it be survivable itself. The SMS should continue operation near total system failure. It cannot rely on any individual component regardless of its level of availability. This requires that survivability emerge from every component through the action of the SMS. The SMS augments every entity (service or physical resource) in an Object Service Architecture (OSA) with the ability to act in its own best interest.

This approach to survivability is to be differentiated from hardening individual resources to the point that they could survive a broad class of predicted threats. As a chain is only as strong as its weakest link, so a service is only as strong as its weakest dependency. Regardless of the availability of a map server, if a weather service is not available, a flight cannot be planned. This example points out the fact that the correct servers must be hardened. Even if both servers are hardened, problems arise if they are not hardened to the appropriate degree. Suppose the weather server is by its nature more vulnerable to attack than the map server. Hardening the weather service to an equal degree as the map server requires additional resources.

Individual resources must be hardened to the appropriate degree and in the appropriate balance to yield a survivable system. Maintaining this balance is the purpose of the Survivability Management Service.

4.1. Resources, Services, and Threats

The SMS allocates resources to services so as to minimize the potential loss of work from threats to the system. At the most basic level, resources are physical assets such as CPU, disk, memory, and bandwidth. Services built upon base resources may be used as a higher level resource by yet another service. At the highest level are those services used directly by an end user. A threat may compromise any service or resource.

Resources, services, and threats all change dynamically. To be managed effectively they must be closely tracked. This is true regardless of whether they are to be managed by an administrator or an automated service. The SMS maintains resource, service, situation, and threat models. The models consist of both static and dynamic data. The resource model contains all resources available to the system, the availability of those resources, and their utilization. The service model includes the information maintained in type and implementation repositories and adds service instantiations and dependencies between services and their instances. The situation model defines objective functions for survivability based on real-world operational requirements. The threat model contains the best estimate of the threats a system faces and their probabilities of success as well as any threats that are in the process of materializing.

The purpose of the SMS is to optimize utility and survivability given the information in the models. This goal begs several questions: what is utility, what is survivability, and

what is optimal. The answer to each depends on the semantics of the applications in the system and the goals of the system's users. Given application specific objective functions, SMS optimizes the system toward these goals.

4.2. Design

The SMS operates directly from the models. Other agents such as administrators or failure detectors update the models to reflect the current state of the system. This separation creates a very strong abstraction barrier between the SMS and the rest of the system which allows the design of the SMS to be considered in isolation.

4.3. Issues any SMS design must deal with

An SMS design must deal with issues related to security, survivability and distributed load balancing. Issues of particular importance follow:

- **Sensitivity to Inaccurate Threat Assessments** -- An essential input to the SMS is a threat assessment; however, the exact probability of a successful attack is never known *a priori*. The SMS must produce good allocations given only a reasonable approximation of the failure probabilities.
- **Fidelity** -- Fidelity determines how often the SMS runs and at what granularity it allocates resources. Fidelity correlates with overhead. The SMS should be able to operate with only a small increase above existing service binding protocols.
- **Interaction with Security Domains and Enclaves** -- The SMS allocates resources from a pool. The larger and more homogenous this pool, the more efficient the SMS can be. The effect of security domains and enclaves is to partition resources. Whether an SMS can operate efficiently in this environment remains to be proven.
- **Local Authority of Command** -- The SMS cannot violate local authority of command. It should have the potential to efficiently allocate shared resources outside the local authority.
- **Thrashing** -- The SMS must exhibit stable resource allocations. Constant reallocation for a negligible performance improvement must be avoided.
- **Partitioning** -- The SMS cannot rely on any central authority that could be made unavailable by a network partition.

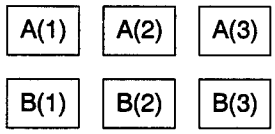
4.4. SMS Market

There are several metaphors for the design of an SMS. It could be considered to be a planning problem with the goals of utility and survivability. Alternatively, it could be viewed as a constraint satisfaction problem. A third possibility is to view survivability as a resource allocation problem to which free market principles apply. The free market approach has the desirable property that optimality arises from the action of every participant in the system acting in its own self interest which satisfies a high level requirement of the SMS.

Appendix A-4

The following example illustrates emergent survivability from a simple computational economy. Suppose there are two services A and B each with three replicas, one active and two passive.

In a computational economy each service is assigned a value. In this example both A and B have a value of 1000 credits. The value of each replica is a fraction of the value of the entire service. For a simple failure model the value can be computed as follows:



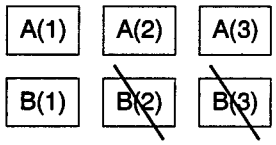
$$\text{Value of Replica} = (\text{Value of Service}) * P(\text{Failure of a Host})^{(\text{Number of Hosts})}(1 - P(\text{Failure of New Host}))$$

The value of each replica is its contribution to the value of the service. That contribution is the value of the service times the probability that the replica will complete the service. Since the replica will only be used if all prior replicas fail, the probability that a replica completes a service is the probability that all prior hosts fail times the probability that this replica succeeds. The value of each replica in the example can be computed assuming that the probability of failure for some time unit is 10%. The first replica for both A and B is worth 1000*(1 - 0.1) or 900 credits. The second replica is worth 1000*(0.1)(1 - 0.1) or 90 credits and the third is worth 9 credits.

The survivability of the entire system can be computed as the probability that all essential services complete. Both A and B are essential services. The probability that either fails is the probability that each of their replicas fail. In general we do not assume independent failure. However, for the purpose of the example assuming independent failure with three replicas, the probability of failure for A is 0.001. Likewise, the probability of failure for B is 0.001. The probability that either fails is 0.00199. Thus the survivability of the system is 0.99801.

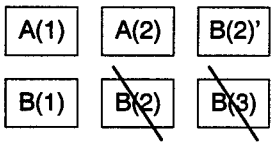
Now suppose there is a concerted attack against B which causes two of B's replicas to fail.

The survivability of the system drops to 0.8991 even though A is operating at full strength. Since B is an essential service its weakness drags the survivability of the entire system down to below its own level. The remedy is to share the available resources between A and B through the computational market.



B is willing to pay 90 credits for a second replica at this point while A is still only willing to pay 9 credits for its third replica. If the host of A(3) receives a bid of 90 credits for its services, acting in its own best interest it should accept and start running a second replica for B.

This reorganization occurs without any global coordination. Afterwards the survivability of the system rises up to 0.9801. Not as high as the initial configuration, but significantly higher than the degraded state without the computational economy.



4.5. Issues specific to a market based SMS

The use of a market in the design of the SMS introduces the following new issues:

- **Non-Deterministic Behavior** -- Given the physical distribution of the resources managed by the market, true non-determinism is introduced into the system. Some

non-determinism is not problematic. The question is whether or not the non-determinism can be reasonably bounded.

- **Market Crashes** -- Market volatility is not problematic if it reflects volatility in either resources or demand. An effective market must manage excess volatility and avoid speculative bubbles.
- **Inflation** -- Inflation is the result of an imbalance between resources and currency. A protocol must be developed to add new currency in proportion to new resources.
- **Local Optima** -- A market is capable of reaching Pareto optimality, a local optima. If this is not sufficient, the market must be augmented with randomization or linear programming to reach a global optima.

4.6. Auction Protocol

A contract is structured as a rate of payment for a specified quality and quantity of service. Both quantity and quality are abstract with respect to the market which brokers any commodity. The rate of payment may include a penalty for breaching the contract. In this way the market implements soft real time constraints. A contract exists for a specified duration. If the server cannot deliver the service as specified in the contract it must pay the breach penalty.

Each client maintains a list of the quantity of each service it requires. Periodically the client checks this list against the services for which it has contracts. For those services without contracts, it requests a list of candidate servers from the EBS. The candidates provide the required service but may do so with different QoS or at different cost. The cost of a service is estimated to be the service's earnings from previous time periods which are kept in a separate earnings database. The servers are further differentiated by their failure characteristics which are kept in the threat model. All this information is taken into consideration by the client broker.

The client broker then writes several contracts, one for each server it is interested in. The contracts may be for different rates or durations because the servers are not homogenous and the client my not be willing to pay each the same amount or to commit for the same duration. These contacts are then offered to the servers.

A server broker gathers offers from clients and when it is ready to accept new work chooses an offer to accept and acknowledges that contract. If another server has already acknowledged the contract, the acknowledgment will fail. The server broker will then select another contract to acknowledge and repeat the process.

The choice of which contracts to accept is complex. In a general sense it amounts to on-line bin packing. Each server broker aims to maximize its profits over time. This criterion does not reduce to any simple objective function like accept the highest priced contract or maximize resource utilization. Both the pending offers and the contracts in effect at the time determine the pool from which the server broker can choose to schedule work. The existing contracts are included because they can be postponed or if necessary breached. Breaching a contract will usually incur a breach penalty which must be paid by the server broker, so only if a new contract will pay enough to compensate for the lost revenue and the breach penalty will it be considered.

Accepting a breach penalty is in itself an issue the server broker must consider. A client that absolutely needs to have a service performed is likely to be willing to pay a relatively high price for that service; however, the client will pass on a high penalty for non performance in the form of a breach penalty.

After a server satisfies a contract, it bills the client. Each resource maintains its own account locally. There is no central bank because this would be a centralized point of failure. The amount is debited from the client and credited to the server. This transaction is not validated, but all actions are recorded in a ledger so if there is any fraud it can be detected off line.

The bidding logic for a new replica is very much as it was described above. The example in section 4.4 assumes that each host has a fixed, independent failure probability. The Threat Model actually used captures the correlation between hosts which the broker for a replicated service uses. At a high level the value that each host brings to the replica group is the same. It is the value of the service times the probability that the prior hosts fail times the probability that the host under consideration succeeds.

5. Models

The Survivability Service maintains three models. The purpose of these are discussed in *Composition Model for Object Services Architectures*. Since that was written, we have combined the Attack and Threat Models into a single Threat Model and made the Service Model separate.

This section discusses the models as they are currently defined.

5.1. Resource Model

The current design of this model is presented in section 7.1 of *Composition Model for Object services Architectures* and is not repeated here. The model as defined is more complex than is used in the demonstration prototype at present. It is not clear at this point how detailed the model must be in order to allow matching of code images to physical resources or how quality of service specifications should be supported.

5.2. Threat Model

The Threat Model models the kinds of attacks that can be launched (intentionally or otherwise) against the resources of the system, the resources affected by such attacks, and the likelihood of an attack occurring and if it occurs, of causing damage.

The Threat Model is used by the SMS to avoid creating brittle configurations that are likely to be easily damaged by probable errors. While the Threat Model is currently used by the SMS market mechanisms, a similar model would be required regardless of the mechanism used by the SMS to determine desirable, non-brittle configurations.

The model was designed to use a single, simple formulation for attacks that affected single resources and attacks that affect multiple resources. The need to treat attacks that might cause multiple failures is important. Because our concern is with configurations rather than single resources, we care far more about whether a configuration composed of several resources survives than that any individual resource survives. This is the basis for

survivability; that it is possible to create and evolve survivable configurations from non-survivable components using redundancy, migration, and clever rebinding. Thus we need to be able to estimate the probability that the configuration will fail.

The following example motivate the need for considering correlated failures. Often, redundant (not necessarily identical) instances of some resource are used. In this case, only some quorum of these resources must survive in order for the resource group to survive. Consider an attack such as a power failure that could cause a host to fail with probability p. The expected number of replicas to fail is np. However, we don't care about how many replicas fail, only whether at least one remains running. For this correlation matters. If all n replicas are allocated on hosts on different power supplies, the failures are uncorrelated and the probability that all replicas are lost is p^n. If for some reason all replicas are allocated on hosts on the same power supply, the failures are correlated and the probability that all will fail is simply p.

The model identifies each kind of *threat* to system resources. Each threat can affect one or more collections of resources called *resource groups*. The realization of a threat against a given resource group is called an *attack* (whether intentional or not). A given resource may be in more than one resource group (i.e., subject to more than one kind of threat). The probability a particular attack will occur is designated $p(group)$, the probability that given group will be attacked in the given way.

So far, this says nothing about whether the attack actually damages any of the resources in the resource group, only that the attack is made. While some attacks against a resource group will definitely affect all members of the group (e.g., the power failure example), other kinds of attacks may or may not affect any given member. However, the failures of such resources because of such attacks are correlated more than if the attacks were independent. As an example of this situation, consider a viral attack against all machines running a particular operating system. Assume there are two such attacks, one targeted at Win NT machines and another at Solaris machines and that both are considered equally likely to occur. Even though the probability of any given machine crashing is the same, it is more likely that two WinNT machines or two Solaris machines will crash than that one of each will crash. To model this situation, for each attack we designate the probability that any given resource in the attacked resource group fails as $p(i|group)$.

The above is illustrated by the following ER diagram. Its implementation in a relational DBMS is straightforward.

Appendix A-4

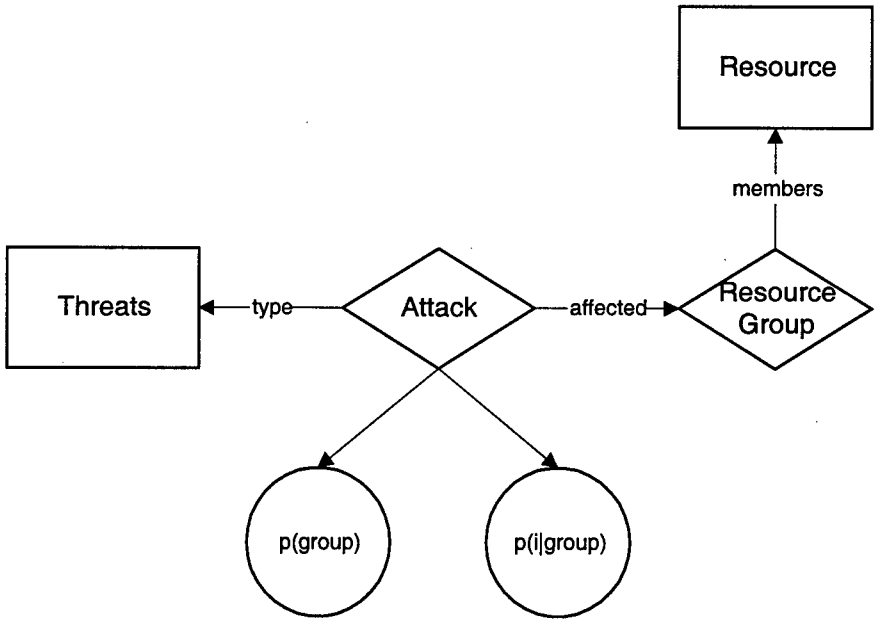


Figure: Threat Model ER Diagram

From this, it is possible to compute the probability that any given combination of resources will fail to retain a quorum in the face of all threats to the system. At present, we use an iterative algorithm to perform this computation; a closed form may be possible.

Two obvious concerns with the above described model are the accuracy with which probabilities can be assigned to intrinsically unknowable events, and the time scale over which probabilities should be measured. The latter is considerably less critical than the former, but it is important to choose a time interval in which single attacks of a given type against a particular resource group can be assumed; in other words, we need to model this as a Poisson process. Sensitivity to modeling error is in itself a research topic; we are seeking funding to separately address this issue. It is our hope that being close is good enough to cause reasonable allocation decisions to be made; remember that the Threat Model is used only to rank the desirability of configurations already determined to be legitimate.

5.3. Service Model

This model will include details about the semantic and performance capabilities provided by individual service instances under different resource assumptions. It should be made compatible with whatever the OMG Trader Service does. This is not an issue until service renegotiation is addressed in the next year. Details TBD.

5.4. Situation Model

This model assigns valuations to the various services based on real-world operational objectives. It may also be used to switch between different Threat Models that reflect differing threat environments based on the real-world situation. Details are TBD.

Appendix A-4

5.5. Using the Models

Once a categorization has been made by the detector, there are some additional decisions about the source and nature of the attack that need to be made. These are outside the bounds of both the detector and the evolution system, but are key to choosing a good evolution plan to allow the system to survive. Among the determinations that must be made are:

- is the problem likely to be temporary or permanent? In this context, permanent also means that it will exist for an unacceptable duration given the needs of the application.
- is the problem an instance, implementation, or protocol failure?
- did the attack originate local or remote to the damaged site? The response may be different depending on status of the site.

5.6. Model Refinement

The models will be changed for a variety of reasons by several different agents. This discussion centers on changes to model content, not schema.

The Resource Model is modified when resources are added or removed, when the failure detectors determine that some resource has become unavailable or degraded, when a resource is assigned (successfully) to some task, and when the implementation ORB reports on status. All are modified directly by the component noticing the change.

The Threat Model is refined when new threats are determined to exist, when a resource becomes part of a different "threatened group" and is exposed to different threats, when threat intensity changes, or when evidence indicates that a previous threat assessment was inaccurate. Changes to this model are likely to be by a human "survivability operator" rather than automatic. We envision this kind of modification taking place from a "security anchor desk".

The Service Model is modified when new services are defined (as part of the interface specification in the survivable OSA abstraction) or when new implementations classes for the services are defined (as part of the interface specification in the ORB-level object abstraction). Either level specification may be modified. A service instance may acquire new capabilities if it learns some new information ("I now have maps of Albania"). An implementation may be provided with new QoS specifications for different configurations (e.g., the performance numbers on a 800MHz Pentium with 1GB of 10nsec RAM would probably not be part of today's specification).

The Situation Model is expected to be quite static. Existing situations are unlikely to be changed. Additional situations are likely to be added infrequently. This is because the definition of an operational situation is a policy decision typically made by humans after careful consideration of operational objectives and will interact with other military doctrine.

Appendix A-4

6. External Components

The Survivability Service interacts with two external components: one or more *Implementation ORBs* and the *Failure Detectors and Performance Monitors*. The capabilities of these are beyond our control. Further, since we want to use a variety of such mechanisms, we cannot even fix the capabilities exactly. For example, while the CORBA standard specifies the form of a binding statement (to connect a client to a service), the interpretation of the argument specifying the service to be bound is left as an implementation decision to the ORB vendors. This argument typically can be an incomplete specification of a service instance's OID, which allows the ORB to choose a convenient (e.g., nearby, already loaded, idle) instance to bind, and provides a degree of portability. Needless to say, not all ORB vendors perform this selection in the same way. Since placement is very important to survivability, this under-specification matters.

This section sketches the capabilities we expect to be provided by any reasonable choice for these two systems. The capabilities listed effectively represent the centroid of the capabilities of a number of similar systems, not the capabilities of any one system.

6.1. The Implementation ORB

TBD. All relevant ORB hooks appear to be provided by CORBA Naming Service, Implementation Repository, Loaders, and the open() binding call.

6.2. Failure Detectors & Performance Monitors

All detectors maintain a model of what constitutes normal behavior and look for deviations from that behavior. Some failure types are generic and can be built into the detectors directly (e.g., failed processes). Other failure types are service-specific and rely on specification of the likely failure modes of the service (from extended interface definitions for fault tolerance).

Detectors can be categorized:

- *Passive detectors* look at patterns of behavior in the system they are monitoring. For example, they may look for unusual communications patterns or file openings. Deviation from that pattern may indicate that a compromise has occurred and that an attacker is attempting to find out what can be accessed.

- *Active detectors* probe the system, sending out messages to see what response will be obtained; they are better able to detect failure (as opposed to compromise) than passive detectors. Wrappers that send out "heartbeats" are examples of such detectors, as are probes like "ping".

- *Reactive detectors* are invoked directly by applications when specified application events occur. The Orange Book requires "certified" applications be able to log such events, which could also be provided to the reactive detector. Often, "failed" events are more interesting than "successful" events, since they have a higher probability of reflecting compromised or degraded behavior. Which events are actually logged is determined in an "accreditation process" that takes into account the risk/benefit of logging or not logging particular events in a given application context.

- *Self detectors* are services that signal exceptions up the calling chain. Such exceptions are often very accurate, but of course only work when the service is still active and connected. The exceptions that can be signaled in this way are part of conventional IDL interface specifications.
- *Manual detectors* are human entered observations and conclusions. They often relate to gross failures (e.g., physical losses) or are the results of analysis of detected events to determine a root cause (e.g., by a CERT-like organization).

Regardless of which type of detector is present, the detector is unlikely to provide a lot of information about why the failure occurred. This is because all detectors essentially look at *symptoms* of attacks rather than the attacks themselves (since this is all they can really see). This has advantages and disadvantages. The obvious disadvantage is that because the detector cannot provide details about the attack that caused the failure, the choice of remedy is less accurate than would be the case if it were known how the system was compromised. For example, without knowledge of the attack, the system might be reconfigured in such a way that it is still vulnerable to the same attack. On the other hand, by looking only at symptoms, the detector can detect the results of attacks that have never been seen before as long as they produce symptoms that fall outside the normal range of behavior.

Detectors are likely to categorize the kinds of detectable events that we wish to respond to as:

- loss of site
- loss of service (other services at site running)
- loss of access to site
- loss of access to service (can get to other services at site)
- degraded performance of site, service, or network (may not be able to differentiate)
- possible corruption of site or service (still providing some useable, but perhaps tainted, service)

These events seem to all be distinguishable using a combination of active, passive, and reactive detectors, and the responses to them are potentially very different. In particular, a major difference between loss of access and loss of site or service is that with loss of access, if access can be re-established, the service still retains its state, whereas if the service or site has been lost, the service and its clients must be re-synchronized, which is not always possible.

Note that in an environment where increasingly a site (processor) is dedicated to a single service, the distinction between loss of site and loss of service blurs, but that even in this situation there is a useful distinction to be made because a daemon running alongside a service can be used to differentiate between loss of the service and loss of access (though not between loss of access and loss of site). Since loss of access and loss of service can have different recovery techniques, the distinction is significant.

Mis-categorization by the detector is certainly possible. Since this is the case, the detector should be biased to place the error into a more severe category (rather than an incorrect category), with the result that either no recovery is possible or a less desirable survival strategy is pursued. Detection and categorization can also be delayed. Thus it is important that synchronicity or failure and detection not be assumed by the SMS.

For the immediate future, we are assuming a timely and accurate failure detector. An important future issue for the SMS is its sensitivity to mis-categorization of failures and strategies for compensating.

7. Related Work

SS focuses on delivering high quality at the service level. This requires utilizing recent accomplishments in the areas of Network QoS, High Availability, Failure detection and market based resource allocation.

7.1 MARX

The Michigan Internet AuctionBot generates an efficient price for any commodity. Its goal is to find a price that is fair to both the buyer and the seller that clears the market. Research into applying the AuctionBot to network bandwidth allocation is being conducted. This work is relevant to the survivability service, but it is not clear whether the AuctonBot can be used directly. The goal of the AuctionBot project is an open, agoric system. We believe that similar functionality can be achieved with lower overhead than is required in an agoric system.

Michael P. Wellman, Jeffrey K MacKie-Mason, Sugih Jamin, "Market Based Adaptive Architectures for Information Survivability,".

7.2 AQuA

In the areas of High Availability and Network QoS, QuO from the AQuA project includes the state of the art. At its lowest levels it is built upon the reliable multicast of the third generation of the Isis project, Electra. AQuA builds reliable, predictable message delivery into a CORBA Object Request Broker. Quality of Service is specified by a high level analogue to IDL called QDL. This distinction separates the semantics of the service and its interface from its bandwidth and timing constraints.

The AQuA system adds network QoS to CORBA so that an application written on a LAN can be used in the less hospitable environment of a WAN. AQuA is a viable component of the Survivability Architecture but it does not manage resources on hosts or high level resources synthesized by services. AQuA passes network QoS capabilities up to CORBA in a transparent manner. Other QoS issues and the resource balancing required for emergent survivability would have to be managed by other components.

John A. Zinky, David E. Bracken, and Richard D. Schantz, "Architectural Support for Quality of Service for CORBA Objects," Theory and Practice of Object Systems, Vol 3(1) 1997.

Appendix A-4

7.3 SMARTS InCharge

This is a system for network failure detection based on resource and failure models. Network components are modeled in an extended IDL which supports the definition of failure modes and visible symptoms of those failures. The system is CORBA-based and models networks at various levels of abstraction in a modeling language called NetMate. There is a great potential for using at least some of the same model and the modeling language should be considered for compatibility. It is possible that their approach to failure classification based on creating code books mapping symptoms to errors would be appropriate for the external failure detectors, but this is outside of our concern.

S. Yemini, S. Kliger, E. Mozes, "High Speed & Robust Event Correlation", System Management ARTS (SMARTS) 14 Mamaroneck Avenue, White Plains NY 10601 kliger, eyal, yemini@smarts.com

Appendix A-5

QoS & Survivability

David Wells
Object Services and Consulting, Inc.

March 1998
Revised August 1998

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

1 - Introduction

In the past several years, there has been considerable research in the areas of *quality of service (QoS)* and *survivability* in an attempt to facilitate the construction of large software systems that behave properly under a wide range of operating conditions and degrade gracefully when outside this range. From the 10,000 foot level, quality of service addresses the goodness and timeliness of results delivered to a client, while survivability addresses how to repair or gracefully degrade when things go awry and the desired behavior is not able to be maintained. These two areas are obviously related, because QoS forms at least a part of the definition of the "desired" behavior of a system that survivability techniques are attempting to preserve or gracefully degrade.

This paper explores the relationship between quality of service and survivability. Section 2 discusses the concepts of quality of service and survivability. Section 3 identifies and presents highlights of important QoS research efforts. Section 4 discusses these projects in more detail, particularly efforts whose approach to QoS is compatible to our approach to survivability. Section 5 identifies technical "points of intersection" between the QoS and survivability work that could eventually lead to a confluence. Section 6 identifies some issues that arise when QoS and survivability are combined and points out some weaknesses in the way the existing projects add and measure survivability.

2 - Quality of Service & Survivability

The concept of quality of service has traditionally been applied only at the network (and sometimes operating system) level. At that level, QoS deals with issues such as time to delivery, bandwidth, jitter, and error rates. Network-level QoS is important because many applications will not function in an acceptable or useful manner unless the network they use can guarantee some minimal service guarantees. It has been observed that just as all services and applications rely on networks, they also rely on other applications and services and these must also make some QoS guarantees to allow the application to perform correctly.

A pair of short papers from Rome Laboratory describe service-level QoS as a function of *precision* (how much), *accuracy* (how correct), and *timeliness* (does it come when needed). For example, a map may be insufficiently precise (100m instead of 10m resolution), inaccurate (things in the wrong places), or untimely (delivered too late to be useful). Unless all three requirements are met, a client is not getting what it needs and therefore the result is lacking in *QoS*. A *benefit function* is defined for each point in this 3-D space stating the value to the client of receiving that particular QoS. Distance metrics for the argument spaces are application dependent. For example, the distance between "red" and "orange" will be less than the distance between "red" and "blue" in a spectral dimension, but not in a textual dimension. The benefit function is similarly application-specific and may be situation dependent as well.

QoS at any given level of abstraction places QoS requirements on the components providing the QoS. In the example, delivering a 10m map requires the retrieval of a certain number of bits of map data.

A survivable system is one that can repair itself or degrade gracefully to preserve as much critical functionality as possible in the face of attacks and failures. A survivable system needs to be able to switch compatible services in an established connection and substitute acceptable alternatives. It must also be able to dynamically adapt to the threats in its environment to reallocate essential processing to the most robust resources.

QoS and survivability are intricately linked; they are not the same, but neither makes sense without the other.

- Survivability without some notion of what is supposed to be surviving is pointless; the what is provided by QoS metrics.
- QoS "guarantees" that can't be made to survive or adapt under changing conditions are not very useful as guarantees, and could in fact lead to denial of service attacks as opponents bring a system to its knees by degrading QoS and causing the QoS management system to continually add superfluous resources.

---

3 - Overview of QoS Projects

Recent work, much of it funded by DARPA-ITO and administered by Rome Laboratory through the Quorum program, is extending QoS concepts and mechanisms to higher semantic levels to allow the definition, measurement, and control of the quality of service delivered by services and complete applications. There are three major grouping of projects. SRI and BBN each have architectural frameworks and are developing or adapting multiple pieces of technology to fit their frameworks. The BBN and SRI frameworks address different types of QoS needs and do not appear compatible. Several independent projects are developing individual pieces of technology. All projects are administered by Rome Laboratory, which also does some technology development. The groupings and relationships of projects (shown in QoS Projects Map) are given below. The individual projects are described in more detail in Section 4.

BBN Cluster: The BBN cluster consists of three architecture/infrastructure efforts based on a CORBA client server model:

- QuO - Quality Objects

team: BBN - see: papers
- AQuA - Adaptive Quality of Service Availability
  team: BBN, Illinois, Cornell - see: papers
- OIT - Open Implementation Toolkit for Creating Adaptable Distributed Applications
  team: BBN, Illinois - see: papers,

and one application demonstration project:

- DIRM - Dynamic Integrated Resource Management
  team: BBN, Columbia, SMARTS - see: papers

These projects are closely related, and in many ways it is useful to think of them as one large project. QuO developed a general framework for QoS in CORBA that is being refined by AQuA and extended by OIT to address service availability management. There is to be a "production QuO" done under the DIRM project.

SRI: SRI is developing an architecture and scheduling algorithms for the delivery of end-to-end QoS for a data streaming model:

- Adaptive QoS-driven Resource Management for Distributed Real Time Systems
  team: SRI - see: papers,

Independent Projects: These projects are developing modeling and analysis/simulation tools that could be used by a QoS management system to model resources and QoS requirements and to schedule resources. Several of the tools were developed for another purpose and are being adapted to the QoS domain. All have had some relationship with the BBN cluster of projects. Projects are:

- UltraSAN
  team: Illinois - see papers
- QoSME
  team: Columbia
- InCharge/MODEL
  team: SMARTS, Inc. - see papers,
- Horus
  team: Cornell

---

4 - QoS Project Details

BBN Related Projects -QuO, AQuA, OIT, DIRM

The QuO project is developing *Quality Objects* that can manage their own QoS. QuO is integrated with the CORBA architecture, in that most of the work is done by extending client-side and server-side ORB stubs. These "smarter" stubs are generated from an extension to IDL called QDL that allows specifying things about service and connection quality. QuO assumes a CORBA-like processing model in which there are client-server and peer-peer relationships and in which the exact processing loads are unknown and can be quite variable. This distinguishes QuO from the SRI work, where the information flow of the applications takes the shape of a DAG and where processing and QoS requirements are assumed to be well understood a priori as is the case in multi-media delivery.

Using QDL, an object can specify the QoS it expects from each service that it uses and can specify what its usage patterns will be (e.g., invocations/sec). An object will similarly use QDL to specify the QoS it knows how to provide (which can be different for different implementations or resource levels). These specifications are used to create client-server bindings called *connections*. Connections are first-class objects as they are in our survivability model defined in Composition Model for OSAs.

To make the writing of QoS specifications and the creation and maintenance of connections tractable, QoS is partitioned into *regions* of normative behavior. Within each region it is assumed that every QoS is equally useable. Region definitions look like predicates in a language like C. Some of these regions (e.g., Normal, Overload, InsufficientResources, Idle) are predefined, but others can be defined using QDL. Clients and services parameterize these regions to define when they are in a particular one. For example, a client may say that when in Normal mode, it will make between 2 and 10 calls/sec to a particular service; anything over 10 calls/sec puts the client into Overload mode.

The use of regions means that minor (insignificant) deviations in the QoS delivered do not require changes to the service or connection, which substantially simplifies runtime processing. Similarly, if clients and servers agree on common meanings for named regions, matching client and server specifications is simplified which is important since this activity must often be done on the fly. The use of QoS regions is a significant difference between the BBN and SRI approaches; in the SRI approach, the benefit function is allowed to be continuous.

The regions discussed so far are called *negotiated regions* since they represent where the client and server try to operate and form the basis for the connection contract. As long as both the client and server operate in their negotiated regions, all is well. However, it is possible for either the client or the server (which for these purposes also includes the connection between the proxy and the remote server implementation) to deviate from the negotiated region, either by overloading the server or failing to deliver results as required. Because of the potential for operating outside the negotiated regions, QuO defines *reality regions* to represent the actual QoS-relevant behaviors of client and servers. Reality regions are defined in the same way as negotiated regions; it appears that for any given connection, the same set of specifications will be used for both kinds of regions. If the observed reality region differs from the negotiated region, the connection is no longer valid and remedial action must be taken.

Various monitors determine the reality region a connection is actually in. Monitors are presumably predefined to monitor the kinds of things that QuO cares about; others can presumably be defined, and there is a claim that they can be deployed selectively to only monitor those QoS items of interest. Types of monitors include counting invocations, timers, and heartbeats. It seems that there could be a whole subsystem for inserting probes in useful places.

Operationally, a QuO object is a *distributed object* where part of it lies on the server side and part(s) lie with its various clients. It is appropriate to think of the client side as being a very smart proxy object that knows how to do QoS related actions. (It appears that this stub can be further tailored to do just about anything, but that doesn't look like a good idea unless the architecture tells what kinds of things *should* be done in the proxy.) A QuO proxy keeps track of the negotiated QoS mode of the connection; interacts with monitors; provides feedback to the client through interfaces generated from the QDL; and takes certain actions to maintain the negotiated QoS.

Client-side proxies can take actions to maintain a QoS mode or change the negotiated QoS region. A client makes a service request through a client-side proxy which decides what to do with method

invocations. This could include sending the requests to a server, servicing them from a cache, routing to replicas, routing to a varying set of implementation objects, or ignoring them and signaling an exception. How many of these they actually plan to provide is unknown, but this is definitely where they fit. It fits very well with our survivability model, except that we would extract this from the purview of the client and move it to a Surivability Service to handle competing service demands. In an ideal world, there would be a large number of generic actions that could be taken by *any* proxy to maintain QoS or shift regions gracefully when it detects a change in client behavior. Our Evolution Model for OSAs defines many ways to evolve configurations. Some of the actions a client-side proxy can take to maintain a region or change regions are:

- A server may be able to shift its implementation in order to stay in the same region without disturbing the client. An example given is to change implementations to trade bandwidth for processing power as resource availability changes. Since the server stays in the same negotiated region, the client doesn't need to see any change.
- The client may request a different negotiated region. For example, it may go idle and negotiate a lower QoS region. It is also possible that a client is detected to have entered a different region. For example, the regions may define Idle as no messages of any kind for 5 minutes. If the reality region enters Idle, it can be treated as if the client signaled that it was going Idle. Thus a client can change modes without having to be implemented for QoS.
- A server may have multiple implementation strategies that will allow it to enter different regions. For example, if the client goes Idle, the server may scale back its own resources, while if a Client enters an Overload mode, the server may add resources or shift to a different implementation.
- The proxy may make an up-call to the client to find out what to do.

Feedback to the client is done by callbacks generated from the QDL. These form an additional interface to the client that is used only for QoS purposes. The proxy can notify the client that the reality region has deviated from the negotiated region. The client must provide *handlers* for these callbacks to either begin a negotiation process with the proxy or to change its own behavior in some way (e.g., slow down, accept lower precision, etc.). Again, it appears that there is a wealth of opportunity to do good things here, but it is not clear how many of them are actually defined.

The QuO papers give a fairly detailed figure of the form of the proxy objects. The partitioning of function seems good and would allow us to extend if we needed to. This looks like where a lot of their work went.

A weakness of the QuO project seems to be that they appear to have a sort of "one-level" service model, in which clients call services which are leaves. Also, the allocation of resources appears to be under the control of the proxies, without much regard for other demands. They briefly mention a language, Resource Definition Language (RDL) for specifying resources and it would be reasonable that there be some sort of auction or scheduler that gave resources out in a non-object centric way, but this gets little attention. It may be possible to handle both multiple levels of objects and sensible allocation for competing resources, but they don't seem to do it. It appears that the architecture and models do make this possible.

AQuA is improving the QuO proxy architecture, attaching RSVP to allow QuO proxies to control QOS over CORBA, and adding regions to deal with availability as well as QoS. Not much published, so what follows is my supposition as to how this will work. It appears that they plan to predefine a number of negotiated regions for availability, where the region predicate has to do with things like

how many replicas will be required in order to stay in that region. Hooks down to a replication manager (probably via Horus or using Electra/Ensemble) are used as the monitors to detect whether the reality region for availability has changed (e.g., a replica died). Actions in response to an availability change would be to start another replica or inform the client of the change and let the client decide what to do in the same way it decides what to do about QoS region changes. UltraSAN is being used (or at least considered) as a way to determine the availability region predicates. Using UltraSAN (and possibly SMARTS/Model), they will model various configurations and use the UltraSAN simulation and analysis tools to determine how long a configuration is likely to stay alive or how long it will take to reconfigure (I haven't actually seen the latter discussed at all, so maybe they do not plan to do this). They will define and analyze configurations until they find one that has the right predicted availability for a particular client need. That will then form a contract for a specific availability region.

It is not clear that they maintain a connection between QoS and availability. It looks like region predicates express both QoS and availability concerns in the same predicate. Since increased availability can degrade QoS (slower with more replicas depending on the operation), this needs to be addressed. It does look like they will be able to determine that the combined reality region does not match the combined negotiated region, but that will not help with the act of finding a good combined region definition that can actually be instantiated.

OIT is just getting started and not much has been written about it. It looks like they will be cleaning up some QuO internal architecture and perhaps developing a toolkit to help write and manage contracts. This may integrate with the use of UltraSAN; it does not say so, but that would not be an unreasonable part of such a toolkit. It also appears that OIT will be using these extensions to provide some support for survivability along the lines of AQuA; it is not clear what the relationship is between them.

DIRM has much more the feel of a technology demonstration than the other projects. Little is written about it, but it appears that the idea is to show off some QoS concepts in a collaborative decision process in some sort of military command post setting.

SRI

SRI is working on end-to-end QoS, with particular emphasis on managing data streaming within a DAG of processes. The strengths of the work is that it has a very clear modeling methodology and language (including pictorial) for specifying alternative implementations, natural handling of information and processing flows that encompass arbitrary numbers of steps, and a scheduling algorithm for allocating processes and communication to available resources as described in a system model.

Resources are allocated by a scheduler based on a modified Dykstra shortest path algorithm for finding least cost (w.r.t. a definable cost function such as least time, cheapest, or throughput) path through a graph representing the required processing steps and communication. There is a delta form of the algorithm that gives delta-suboptimal decisions much faster. A concern is algorithm speed and the fact that it appears to require a centralized scheduler.

It appears that the scheduling algorithm precludes the use of this work in a peer-peer or client-server setting; if so, this is a major differentiator between the SRI and BBN work. There are several reasons for this. It is hard to see how a shortest path algorithm can be adapted to handle arbitrary numbers of

loop iterations as can be found in general client-server or peer-peer systems. A feature of the algorithms is that processing steps and flow be able to be identified on a time step basis; this doesn't seem to match an environment in which service requests are either random or have high variance. All their examples deal with media delivery. The papers mention the ability to separate feedback paths from feed forward paths, but this is not explained.

They define more completely than anyone else surveyed the meaning of precision, accuracy, and timeliness, specifying explicit parameters for these concepts. Each parameter has several components, including absolute values, relationships between values, expected values, bounds, and variance. It looks like a QoS specification could become quite complex. They also define a *benefit function* in the obvious way. It is not clear how their scheduling algorithm deals with relative values or variance that their QoS specifications allow.

This work appears most suitable for rather tightly controlled situations where a high level of QoS is required. Unlike the BBN work, their solution appears to not be open.

UltraSAN

UltraSAN is a system for modeling and analyzing networks in which events such as workload and failures are probabilistic. A system is modeled as a stochastic activity network (SAN), which is an extension of a Petri net. A SAN extends Petri nets by allowing transitions to be probabilistic, for multiple tokens to be at any given place, and "gates" to act as predicates to define which transitions are allowed at any given time. The result looks like a stylized dataflow graph of the activity under analysis. Reward functions are given for transitions to designated states and for remaining in a state for a duration. The point of framing the system as a SAN is that a SAN can be converted to a Markov model, to which they know how to apply analytical and simulation techniques. They have tools to define the SAN and to convert it to a Markov model and analyze & simulate (by means of fault injection) the resultant Markov model. They also have some partitioning and replication constructs to aid in the construction of SANs. Basically, you can "join" SANs together so that they share "places". This allows replicas to be composed and for large SANs to be created from small SANs. They also have a technique based on these replication tricks to reduce the state space of the Markov model, which otherwise would quickly become unmanageable for reasonable sized SANs. The software is currently distributed to 129 universities & several businesses

AQuA is attempting to apply UltraSAN to availability management by building a SAN for a projected configuration (along with the reward functions), converting it to a Markov model, and solving to determine what level of availability it can provide. I don't see how this could be done on the fly, so I presume they intend to do off-line evaluations of various configurations they think they can construct, rate each as being survivable or available to some degree, and then define "regions" around these precomputed configurations. e.g., UltraSAN predicts that under a given set of transition assumptions, 3 replicas gives availability of 0.9999 for T seconds, so a region of high availability would be that 3 replicas are maintained. Failure to do so constitutes a "reality" region change, which requires reconfiguration or change to a different "negotiated" region. I don't think this can deal well with configurations not previously planned as desirable and not with changes to the transition parameters as would happen if the system came under attack. This might allow a degree of "preplanned" survivability, but does not appear highly adaptive.

None of the papers directly talk about evaluating configurations for survivability under exceptional conditions. All deal with expected behavior of a configuration to determine the "reward" from a

particular configuration.

Limitations on the use of UltraSAN in AQuA or OIT appear to be:

- Modeling in UltraSAN looks quite complex, so not a lot of alternative configurations will be able to be tried. Hence, they will not be able to employ as many survivability techniques as they might actually have available.
- UltraSAN analysis techniques are heavy weight, since they require repeated simulation or solution of huge Markov models. This means that all analysis will have to be done in advance; i.e., when contract regions are established, not when something breaks. This has the same disadvantage as above.
- UltraSAN models do not appear to be easily modified if the interconnect topology changes. This is not a big problem in the original target environment of UltraSAN, where topology was based on physical interconnects, but is more of an issue in a service model, where the topology changes as easily as starting a new process.
- It is not clear how UltraSAN deals with a time-varying mix of tasks and loads.

Columbia - QoSME

Columbia University has done work on QoS for stream data. It is not clear how this work differs from RSVP which is more mainstream (I haven't investigated enough to judge at any deep level). Possibly the emphasis in this project is on scheduling and resource allocation rather than mechanisms.

SMARTS - MODEL & inCharge

MODEL uses the NetMate resource model to describe systems. MODEL does fault isolation by treating symptoms of faults as bit errors in a code and then using error correcting techniques to isolate the "code" (the original fault) that caused the "received" symptom code. It is an elegant approach and is claimed to be fast. A problem is that the connection topology is rigid, and since this determines the code book, component reconfiguration (sw or hw) will force reconstruction of the code book. Also, since symptoms appear over time, it is not clear how to assemble them into a "code" that can be decoded. The difficulty is figuring out an appropriate time window, particularly when symptom reports may be delayed or lost. The NetMate model could be a basis for our resource and application models.

Cornell - Horus

Horus is a group message delivery and synchronization system for networking. It is a direct descendant of the ISIS system. Horus has been used to manage replica groups as part of making individual services highly available through controlled redundancy. Horus is not currently licensable.

5 - Commonalities Between QoS and Survivability Techniques

As should be obvious from the previous sections, there are a number of similarities in architecture, mechanisms, and metrics between service-level QoS efforts and our survivability work. In this section, we examine these commonalities. The next section discusses differences. The terminology we use comes from CORBA, although the observations are also applicable in other object-based frameworks.

QoS and Survivability Specified by Client; Managed by System

All work in QoS and survivability assumes that while clients are able to specify the value they place on receiving a given QoS or having a service remain available, they should not manage it themselves. There are four reasons for this:

- While managing either of these is hard for a service developer to implement on a per service basis, the techniques used are generic enough that they can be developed well by QoS and survivability experts and applied to a wide variety of services and situations.
- QoS and survivability both require resource reservation, which in turn requires substantial knowledge about the resource environment. The environment is the same for all services in it, so it makes more sense to model it outside of any of the individual services. This especially true since the environment will be expected to evolve (for good or ill) over the lifetime of the services.
- Services cannot make unilateral decisions on resource reservation because they have to compete against other services, whose existence may be unknown to them. Every service will naturally attempt to maximize its own behavior, but in resource constrained conditions, this will be impossible. This will require at minimum a non service centric allocation mechanism.
- The relative values of the services themselves will change depending on the situation. Services that are valuable in peacetime may become considerably less valuable during combat. Unless a service is programmed to understand all the operational contexts in which it may find itself and adjust its demands accordingly, this is impossible to achieve. This is unreasonable to expect, especially for services that may be used for many years and be adapted to new contexts. It also does not handle greedy services that chose not to play by the rules, perhaps from malevolent intent.

Moving the locus of QoS and survivability management out of the client requires significant extensions to both binding specifications and the CORBA proxy architecture.

Binding Specifications

A binding specification used in a system managing QoS or survivability has to specify both more and less than current, OID-based bindings do: more because things such as performance, abstract state, trust, availability, and cost of use must be specified in addition to type; and less because if the specification is too specific (e.g., identifies a single object or server) QoS and survivability management will not have enough choices available to do a good job. Once the binding specification becomes more sophisticated, services will be required to advertise their capabilities to a far greater extent than they do now. Binding specifications and advertisements will need to be matched to determine a (possibly ranked) set of (complete or partial) matches. In an OMG context, this would be the job of a Trader Service, although far more sophisticated than any yet existent, even in prototype form. Because trading probably requires domain knowledge to achieve good matches, we will probably see a family of domain-specific Traders. If this is true, then matching a complete binding specification will require consulting several Traders and composing their responses. If this approach is taken, it will be possible for the composition function to weight the various responses (e.g., to care more about the speed of response than the security of the service under some circumstances).

Proxy Architecture Extensions

Since a client should be unconcerned with how QoS or survivability is provided, these must be provided by some other part of the system. Both BBN and OBJS have chosen independently to make the control locus of this the server client-side proxy (the local stand-in for a remote server object) and the server-side stub (which handles communication on the server end of the connection). CORBA proxies and server stubs are generated from IDL specifications and are responsible for message passing between clients and remote server objects via the ORB. This requires argument marshaling and interaction with the underlying communication layer. Standard CORBA proxies hide the location and implementation of services from the client, but do little else.

There are several advantages to extending CORBA proxies and server stubs:

- All communications between clients and servers must, by definition, pass through the proxies and server stubs. Arguments and return results are exposed and packaged for movement at this point as part of the normal argument marshaling. Together, this means that all communications can be mediated and that what is passing through the connection is easily accessible.
- A proxy and the server-side stub reside in different address spaces (and often on different machines). This gives some flexibility as to where a given extension should be placed. This can be an advantage for performance reasons. In addition, since functionality can only be provided reliably if the monitor is incorruptible, having the ability to place monitors and mediators out of harm's way is advantageous. This is especially true for things like security monitoring, where a client or service may wish to avoid the mediation and could attempt to corrupt local monitors; the OMG security Service does this. Finally, because proxies and stubs can fail independently, it is possible to place monitors in each to monitor the health of the other.
- The CORBA specification allows proxies and stubs to be extended to do other tasks besides message passing. Alternative proxies have been used to manage local caches, to distribute processing load among replicas, and to manage security (in the OMG Security Service). Commercial ORBs often provide server stub alternatives that wrap the server implementation via inheritance or delegation.
- Smart proxies can be generated automatically by an improved IDL compiler that either replaces or extends the IDL compiler that is provided with every ORB to generate proxies and server stubs. [Note: replacement is more likely than extension because, although they could be, these compilers are in general not open to extension.] IDL can be extended to allow definition of additional properties as in the case of BBN's QDL for defining QoS attributes.

There is considerable latitude for the internal architecture of extended proxies and server stubs. The primary questions are: what belongs on the client side and what belongs on the server side, whether existing proxies should be extended or used as-is by a more abstract connection object, and the definition of interfaces to monitors, up-calls to clients, and external services such as the Survivability Service.

Adopting BBN terminology, we call the entire collection of mechanisms between a client and a service a *connection*.

Connections

For both QoS and survivability, a connection between a client and a service is far more abstract and far more active than in CORBA.

The need for increased abstraction is because the enhanced binding specifications allow considerably more binding alternatives and consequently more flexibility in the way the connections are established and managed. This additional flexibility makes it important to make not only the location and implementation of the object providing the service hidden from the client, but also which object is providing the service. Because of this, the client must not be allowed to act as it is connected to a specific object providing the service or to expect anything about the service other than what it requests in the binding specification it provides.

The increased activity is because the connection is attempting to guarantee far more things about the interaction than simply "best effort" to deliver requests and return results. It does this by maintaining information about the desired behavior of the connection in the form of a *connection contract*, measuring the actual behavior through a collection of *monitors* attached to the proxy and server stubs, and taking remedial action if the two do not match. Remedial actions include changing the implementation of the object providing service, changing which object provides the service, renegotiating the connection contract, or terminating the connection gracefully.

Because of the above, the connection itself becomes a first-class object and should have interfaces through which its activity can be monitored as well as the interfaces used by the client and server.

Monitors

It is important that monitoring be defined and performed by the QoS or Survivability Service rather than either the client or the server. Part of this is a trust issue; both clients and servers have to adhere to the connection contract and it seems unreasonable to trust either to do so. Another part is that many kinds of monitoring (heartbeats, traffic counting, etc.) can be independent of application semantics and should be factored out. The relative importance of different monitors depends on what properties are defined as important by the connection contract, allowing monitors to be placed only when needed. Finally, monitoring on the connection itself allows "QoS-unaware" and "survivability-unaware" services to have a certain degree of these "added" to them without requiring reimplementation; an important consideration given the number of existing services that are unaware.

6 - Issues in Merging QoS & Survivability

It is tempting to think that QoS and survivability can simply be composed. However, it is not quite so simple as can be seen by the following. The later BBN work attempts to add "availability" onto QoS by controlling a replication factor for the service implementation. This does not seem to capture a number of key points and seems somewhat redundant. Specifically:

- If a service fails to be available, and hence doesn't respond when needed, isn't that a QoS failure? How is the failure to deliver a timely response because of service crash different, from the client perspective, from failure to deliver a timely response due to any other cause that would be covered by QoS considerations? Why should they be specified independently?
- A service that is up 100% of the time but provides low QoS isn't really "available". Just the fact that a service is running is irrelevant.
- To meet a given QoS requirement, there is no requirement that a service be continually available; only that it do what it is supposed to do at the right time. Assume that a service has a QoS requirement of 99% of its responses to be within a 10 second time window from method invocation. As a QoS metric, that is pretty clear. However, when mapped to availability, it is

less obvious and there is not a clear correlation. If a client sends 100 messages/hour and each takes 1 second to process, there is no requirement that the service be "up" 99% of the time. The service will actually be doing useful work for 100/3600 seconds (<3% of the time). So, as long as it can be brought up fast enough when needed, there is really no 99% "availability" requirement. This false issue does not arise if the entire matter is couched as a QoS requirement.

- If the time to start a service was zero and all services could persistently save their current state for free, a service could be activated only and exactly when it had work to do. In this case, there would *never* be a need to have a service running until it was needed, which would obviate the need for replication except for the persistent state. Again, replication and availability is not exactly the right measure.
- High availability can be obtained by the use of many replicas. This is not without overhead for replica coordination, so the result is likely to be reduced performance leading to possibly reduced QoS. Treating QoS and availability separately makes this analysis harder.

I think there are these underlying causes of the above puzzles:

- availability as a client concern
- too few reconfiguration strategies
- QoS not pushed through all levels of abstraction
- inadequate treatment of time
- configuration and QoS "fragility" not considered
- inadequate metrics

Problem: Availability as a Client Concern

The BBN QoS work models the "availability" of a service bound to a connection; in the examples this is done by requiring a particular replication factor for the service as part of the connection contract. This is problematic because availability properly applies to a *service*, while QoS applies to a *connection*. It is reasonable that a service be replicated in order to provide the connection the desired QoS, but the coupling should be much looser than implied in the BBN work. To see why treating service availability as a property of a connection is not correct, consider the following examples, in which the client is satisfied if 99% of requests are handled in a timely fashion with suitable precision and accuracy.

- Case #1 - Service A determines that it must be available 99.99% of the time to satisfy 99% of the requests by a client.
- Case #2 - Service B determines that it can be booted quickly enough that it can be idle until needed and never has to be "available".
- Case #3 - Service C determines that it has enough resources that if available 99.9% of the time it can satisfy 99% of the requests of 10 clients.

It is not clear how any of these reasonable cases can be addressed by a model where service availability is a direct concern of the client.

Problem: To Few Reconfiguration Strategies

The QoS work surveyed has a limited number of ways to reconfigure a service or a connection. This causes a tendency to view QoS problems as client problems (demands too high) or a server problem

(too slow or unavailable) rather than more abstractly as a connection problem that can be addressed in a variety of ways to maintain the contracted QoS. Our *Evolution Model for OSAs* paper enumerates rebinding points where things like platform, platform type, implementation code class, replication factor, replication policy, and bound service instance can be changed. When QoS is introduced, a number of other techniques can be used. For example, consider a QoS contract that requires responses within 30 seconds, but that allows a null response if the last non-null response was within the last 5 minutes. In this case, the proxy is free to substitute a null response to meet the timing constraint in some situations. It is instructive to note that if precision or accuracy is allowed to go to zero (even if only occasionally), arbitrarily good timeliness can be acheived by generating null responses in the proxy. This will allow a synchronous client to continue to function (at least for a while) even if the server doesn't respond. An enumeration of a wider range of QoS-preserving responses would help not only by providing a laundry list of useful techniques, but would help to stear the community away from such "solution-oriented" metrics as availability.

Problem: QoS Requirements Not Pushed Through All Levels of Abstraction

Both QoS and survivability must deal with multiple levels of abstraction. A service can only provide a given level of QoS if the services it relies upon can themselves provide the QoS required of them. Similarly, a service is survivable if the services it relies upon servive or if the service can reconfigure to require different base services. Although both communities realize this, at present, neither handles it particularly well. In particular, problems revolve around mapping requirements at one level of abstraction down to requirements at lower levels of abstraction, and efficiently reserving resources through possibly several layers of lower-level services.

A related difficulty is that the QoS policies at higher layers need to somehow be reflected down to lower layers in meaningful form. Consider this example. A client requires high availability (in the BBN model) and fast response. If availability was the only QoS requirement, the existence of the requisite number of replicas would be sufficient to maintain the contract. However, this is not adequate, since response time also matters. Different replication policies (primary copy, voting, etc.) give different QoS behaviors. Further, the messaging policy within a replica group may vary depending on the QoS needs of the client. For example, normally reads are sent only to a single replica to conserve bandwidth and processing. However, if timely delivery is crucial, it makes sense to attempt reads from multiple replicas to ensure that at least one responds in time (assuming of course that the multiple reads will not compete for the same bandwidth and become even slower). The policy to follow could vary based on a number of factors, including load on the resources, criticality of timely response, client value, and previous behavior of the connection (e.g., is it barely making the timing bound?). To acheive this, it would appear that the high level QoS goal be somehow pushed down to the replication and messaging subsystem.

Problem: Inadequate Treatment of Time

The QoS work surveyed seems to treat all configurations as having present state only. This neglects the fact that configurations will change state (for better or worse) either on demand or because of some random event, and that these changes take some amount of time to occur. In some of these new states, the service will be able to provide the required QoS and in others it will not. Clients promise certain invocation rates as part of the QoS contract, which is part of the treatment of time, but nowhere is the time for a service to change configurations addressed. As noted, this means that services must be kept in a state where they can respond in the QoS bound, which can be very wasteful.

Problem: Configuration & QoS Fragility

QuO treats all responses that fall inside a negotiated region as being of equal worth. That is fine as far as the client is concerned, but is misleading when the ability to meet future QoS goals is considered. Consider the following two contracts whose timeliness components are:

Contract #1

- 10 sec - 30 sec response
- 15 sec average response over any 10 invocation intervals
- no violations allowed

Contract #2

- 10 sec - 30 sec response
- 15 sec average response over any 10 invocation intervals
- no more than 1 timeliness failure per 100 invocations AND no more than 2 average responses over 20 sec per 100
  intervals

With these two contract fragments,:

- in Contract #1, if the average response time is 14.9 seconds, that is not as safe as if it were 11 seconds, since a response time of 30 seconds on the next request would cause a violation in the former case but not the latter
- in Contract #2, if a timing goal has been missed, another timing goal cannot be missed until a certain number of responses have been made

Further, regardless of the previous behavior of the connection:

- if the service configuration fails, there is an increased chance that the next QoS target will not be met; this is influenced by how long it takes to restart the service (if possible)
- if the threat environment becomes more hostile, the probability of missing a QoS target increases, even if no targets have been missed so far
- if load increases, missing QoS targets becomes more likely, even though none have been missed so far

There needs to be some treatment of how likely it is that a negotiated region will be violated in the future. Otherwise, the only time a reconfiguration takes place is when the region *has* been violated, by which time a QoS failure has occurred. Because the time to reconfigure is not treated by the present body of QoS work, it is not even known how long the connection will be unusable. To make matters worse, some failures cannot be recovered from. As part of evaluating how brittle a connection is, there are definitely gradations of "badness", in either probability of a violation, how severe the violation is likely to be, how long it will last, and whether alternative levels of QoS can be reached from the new configuration.

The BBN work partially addresses this problem of brittleness by requiring a given availability for services based on a precomputed replication factor, which is made part of the QoS region contract. By

maintaining replicas, the service never becomes too brittle. While this approach suffers from the limitations discussed above, it brings out an interesting point that we will exploit more fully in our work combining QoS and survivability. In the base QuO work, all parts of the region contract were relevant to the client and all violations were seen by the client (although often they could be fixed by the proxy). However, the replication factor is not relevant to the client except as a rough measure of the ability to deliver *future* QoS. When a contract is violated due to replica failures, only the proxy is concerned. We are working on extending and formalizing this notion that a contract should have parts relevant to both the client and the proxy so that QoS can be delivered and brittle states avoided.

Metrics

Both survivability and QoS attempt to say something about the "goodness" of a connection. To this end, they each define metrics. However, in our opinion, they say very different things. Specifically, QoS (in its broadest sense) addresses current properties of the connection, whereas survivability is concerned with the future of the connection. In other words, survivability addresses whether the connection is likely to maintain a desired QoS. Both QoS work and survivability address restoring a connection to a good QoS; the difference is that when doing so, survivability considers the ability of the new configuration to survive whereas QoS does not. This is discussed in a paper Survivability is Utility.

This has two principle effects.

- The survivability metrics must differ from the QoS metrics, since the latter measure only current state. To give an extreme example of the difference, consider that placing a service on a lightly loaded, but very vulnerable machine will score highly on the QoS metric, but low on the survivability metric. These metrics, and their relationship, are discussed later.
- A Survivability Service will need to take expected future behavior into account when allocating resources. This necessitates some sort of model of likely future events that could cause a configuration to change.

Bibliography of QoS Papers

Papers are organized by project.

Rome Laboratory

- Quality of Service for AWACS Tracking, Patrick Hurley ,Tom Lawrence, Tom Wheeler, Ray Clark, to appear 4th International Command and Control Research and Technology Symposium, 14-16 September 1998, Nasby Park (Stockholm) Sweden
- Anomaly Management, Thomas F. Lawrence, AFRL, internal report, 1997

BBN Cluster Papers

QuO

- Project Overview (1995)
  initial objectives

- Overview of Quality of Service for Distributed Objects (1995)
  subsumed by Architectural Support for Quality of Service for CORBA Objects
- Object-Oriented QoS for C2 Adaptivity and Evolvability (1996)
  overview
- Object-Oriented QoS: Some Research Issues (1996)
  subsumed by Architectural Support for Quality of Service for CORBA Objects
- Architectural Support for Quality of Service for CORBA Objects (1997)
  key paper - defines QuO terminology & outlines the QuO proxy internals
- Specifying and Measuring Quality of Service in Distributed Object Systems (1998)
  key paper - defines the QuO proxy architecture & the CDL contract language

DIRM

- AQuA and DIRM: QuO Projects Overview (1996)
  context example of collaborative planning across a WAN
- 1997 DARPA Project Summary (1997)
  describes integration of QuO + WAN (with RSVP + QoSME) in a test bed

AQuA

- Adaptive Quality of Service for Availability (AQuA) (1997)
  project overview as seen by BBN
- 1997 DARPA Project Summary (1997)
  QuO + Ensemble/Electra + UltraSan; improvements to Ensemble stack, control of replication level of service from QuO
- Adaptive Quality of Service for Availability (AQuA) (1998)
  project overview as seen from Illinois; regions used for setting availability (replication) levels

OIT

- Toolkit for Adaptable Distributed Applications (OIT) (1997)
  description of planned QuO internal refinements; new start

SRI Papers

- QoS Taxonomy (1997)
  good definitions of QoS parameters and benefit function
- Modeling for Adaptive QoS (1997)
  describes a multi-level model for application-specific QoS specification that supports implementation alternatives and variable QoS
- QoS-Based Allocation(1997)
  describes a scheduler for resource allocation in a network that is QoS aware

Illinois Papers

- Dependability Evaluation Using UltraSAN (1993)
  best short introduction
- Specification and Construction of Performability Models (1989)
  detailed introduction, but difficult to follow as a first exposure to the technology

SMARTS Papers

- "High Speed & Robust Event Correlation" - Yemini, Kiger, Mozes, Yemini, Ohsie. Description of the SMARTS inCharge fault analysis system.

Appendix A-6

Notes on Command Post Scenario[1]

David Wells

Object Services and Consulting, Inc.

1. Preface & Caveats

Survivability preserves and gracefully degrades the functionality of software applications. To do this, the system is reconfigured as components fail and situations change. Naturally, not all reconfigurations are legitimate and of those not all improve survivability. To allow us to focus on the kinds events that must be survived and the survival actions that can be taken, we consider survivability in the context of a (admittedly stylized) DoD command post. This context has the advantage that it is relevant and understandable by DoD and the DARPA research community, exposes us to a cross section of real operational concerns, and is potentially insertable without too much effort into larger DoD efforts.

This paper is an amalgam of a number of DoD and contractor documents related to how software and data is (and will be) organized and to command post activities. This report should be read as working notes only.

There is a lot of interpretation on my part, since I have not been able to find a single, comprehensive description of this anywhere. The material was difficult to sort out, and I'm by no means certain that I am right. Part of the problem I think is that there are architectures at several different conceptual levels (DoD-wide or some smaller unit like a JTF), at several different distances into the future (the ultimate goal of fully portable everything and various transition paths from current practice), and differing ties to operational considerations (care or not care about autonomy of command when allocating resources to tasks). Since our objective is to provide survivability mechanisms, I've tried to sort this out into something that is far enough out to be technically interesting, but tied enough to current reality that it is plausible for use in DoD in the next 5-10 years. For instance, since I don't believe that operationally DoD will ever resemble arbitrary meta

---

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

computing with totally fungible resources allocated without regard to a command structure, I've concentrated on the clustering and ownership of resources that I believe will exist into the foreseeable future. The discussion also brings in information I've picked up at various briefings and through reading general military history, which tends to have a lot of side information about how the military is organized.

The paper is organized as follows. Section 2 summarizes the salient points of how DoD wants to have its general computing environment organized in the future. Section 3 discusses what these general software characteristics mean in the context of a command post. Section 4 provides an example of an activity that takes place in a command post (air campaign planning) and how (at a coarse level) it uses the computing resources and how the various parts of the activity interact. These will eventually (in another document) be used to: a) determine survivability requirements and strategies, and b) define a survivability scenario for the command post.

2. Observations on Characteristics of DoD SW & Operations

DoD wants as much portability and fungibility of resources as possible, given operational considerations. They are obviously sick of stovepipes and incompatible solutions. However, I don't think that means they will ever get to either homogeneity or total fungibility. They will always have a big range of machine capabilities because they need data servers, number crunchers, embedded, and portable devices. Further, they need to upgrade and since there is so much HW and SW out there, they can't possibly change it all at once. Also, heterogeneity is good because it helps eliminate catastrophic design or implementation flaws from killing everything. Bottom line for us is that there will be a lot of flexibility of placement and service type, but not unrestricted in either dimension.

DISA TAFIM & TRM talk about a hierarchy of services. They are trying to standardize the base things everyone uses, like OS, main data formats, small number of implementations of a given functionality (and then only to get robustness). I think we can assume that this happens eventually, since it is a big cost savings and improves interoperability in a pretty easy way. At each level down (they talk about 3-4 levels), new services can be added and ones from higher levels can be customized. These then become standard services for that piece of the hierarchy. If sibling areas both customize, they appear to interoperate by the conventions of the parent service; possibly the functionality gets pushed up to the parent level. Note that unlike other organizations, DoD is in the position of being able to enforce its internal standards if it wishes. The further down the hierarchy you get, the less scope there is for customization, but the larger number of customizations there are, since each affects smaller groups (ultimately only one user).

There is a commitment to the Model-View-Controller interface model. There will be a set of standard ways to display data sets on different kinds of devices. This set will be extensible.

There is a distinction between "common", "shared", and "replicated" services and interfaces. "Common" refers to code, interface, or data format commonality; maps will be in a common format and everyone will all use the same spreadsheet. We'd think of

this as being common IDL types. A "shared" service means that the instance is being used from multiple places; a big database in Washington would be shared. A shared service is shared either to ensure consistency for its users or because it requires resources or environment that cannot be replicated. A "replicated" service would be physically replicated code and data. The replicas might be synchronized, but for operational reasons they may be allowed to drift apart by some amount. Examples would be databases of plans or maps, where the cost of synchrony would make them unusable and some discrepancies can be tolerated. The kinds of discrepancies allowed are situational; some differences matter more than others and it is not just volume or time that matters. For example, not knowing exactly how much ammunition you have is OK, but not knowing the current theater boundary is not.

Commanders have missions and have sufficient authority and resources to achieve them (hopefully). A unit may be given different missions, but there is a chunkiness to units that precludes them being taken apart to have portions of themselves reassigned. At a human level, this is for training and morale reasons. Computers are seen as a piece of equipment like an artillery piece and neither will be taken away from their owner.

At each level in a command structure, there are discretionary resources that may be allocated by the commander to any mission of the sub-units. These may be reassigned at will by the commander. Examples are brigade level artillery, which may be used to support any sub-unit of the brigade. This may be used to support an adjacent brigade, but only at the discretion of the commander owning the artillery. It will not be reassigned by someone else. The commander at the next higher level can use some of his discretionary resources or reassign a sub-unit. The higher up the chain you go, the greater the amount of resources and the bigger they tend to be. For example, you might find field artillery at one level and tactical air support somewhere higher up. I would assume this model would hold for computing resources also.

The TAFIM and TRM try to categorize services and support functions. This may lead to some rather arbitrary partitionings, but some categorization seems necessary just so they can comprehend a problem of this size. Efficiency is definitely to be sacrificed in order to get understandability and well delineated functionality so they can determine who is responsible for what. Even if this means that some actions get done redundantly.

Databases in particular follow a model of a big, remote database where ground truth (perceived) is held, with local caches made from it as views. This seems to be pervasive. This is the model we saw back at TI on the FRESH naval operations planner that was eventually deployed in Hawaii. Sometimes the cache is refreshed on a schedule (e.g., once a day) and sometimes it is refreshed on-demand. It is important to realize that no one expects cache consistency to be maintained like you get in a memory cache; it is not needed operationally and is impossible to achieve. They are comfortable working off stale information as long as they know how stale it is and how fast the relevant parts can physically change. The refresh rate is tied to this. The naval planner was refreshed daily, because ships don't move that fast. For urgent information there were side channels for messages.

Appendix A-6

Peers need to exchange some information across boundaries. The military works on regions of responsibility (probably the wrong term, in physical partitioning these are called "theaters" at least at the large scale) and units are expected to keep out of each other's way by staying on their side of the boundary. That way you don't shoot your friend. Limited information has to cross these boundaries; only changes in the boundary conditions or local state that may cause the boundary to get changed in the future. No one needs to know all details of what their peers are doing, only how it is likely to affect them. These boundaries can be exploited.

The DISA SHADE (Shared Data Environment) is an "infosphere". The idea is standard data formats, databases, schemas, meta data, etc., so that commonly used data is managed at a higher level than the producers and consumers. One interesting thing is that they are defining standardized "data segments", which are partitioned data sets for particular uses, along with all the DBMS, scripts, data, schema, and so on so they can be installed wherever they are needed quickly. They already have a few it appears.

SHADE differentiates between "unique" (local and not shared with no particular format or portability requirements), "shared" (used by two or more groups and must have a common schema), and "universal" (used by many groups and managed outside all of them) data sets. SHADE also makes a distinction between databases used operationally and kept up to date but supporting only simple transactions, and warehouses that are culled from operational DBs periodically and against which complex planning operations take place. The warehouse may store raw data or abstracts. SHADE assumes legacy DBs and pairwise communication of data in proprietary format for the foreseeable future. SHADE distinguishes between "distributed DBs" that are kept synchronized, "replicated DBs" that may have temporal differences even though there is a master through which updates are made, and "integrated data servers" that are shared by several applications. I'm not sure what the point of the last one is.

3. Translating the Observations to a Mobile CP

Here's what I think a mobile CP will look like, given the above. We need not simulate all of this, but right now I want context.

3.1. Relationship to Other Units

A mobile CP will need to communicate with its subordinates, with its US or NATO country base (where at least a portion of itself remains), and with peers. SHADE universal data sources will reside outside the CP at its base. SHADE unique data sets will reside in the CP. SHADE shared data sets will either reside in a peer or be shared within the CP.

3.2. Communications

Within a CP, communications is by LAN. There may be multiple LANs with bridges. A LAN may be duplicated for reliability.

There is broadband communication to/from the base, but this may be interrupted or assigned elsewhere. Think of a wide WAN.

Appendix A-6

There is a very wide channel from bases to CPs via satellite that is used for broadcast messages. These are scheduled to correspond to data servers (DS) refresh cycles. This communications path is non-interruptible unless the CP's receiver is down. BADD is the model for this.

Communications to subordinate units is by a WAN by radio or local phone lines. It is lower bandwidth and potentially unreliable.

Communications to peers are by WAN, but are more reliable than to subordinate units because they are not moving and are sited to make communications more reliable.

3.3. Data Management

Operational data is either fed directly from sensors to applications or is stored in a DB local to its consumers. If stored in a DB, it may be accessed by many clients.

Right now, messages from the outside appear to go directly to their clients. There is a move to insert a message DB that stores all messages arriving at the CP. Messages would then be validated and cleaned up by an application before going into the DB. Views of the DB would be used to route messages to the correct consumers, or consumers could query the message DB. In this picture, messages are treated like any other data item, not as some fundamentally different thing like they are now.

Access to externally maintained DBs is through Data Servers (JTF lingo) that convert queries, produce views (possibly by combining and filtering several DBs), and possibly cache. If there is caching, there may or may not be a requirement that the cache and the source DB remain consistent. Whether they are required to be consistent (and how consistent) depends on the needs of the clients and will be specified.

Updates to external DBs appear to go through the Data Server as well, although I'm not sure. At least in the naval planner (FRESH) TI did, the equivalent to a Data Server was read only and the application could never update; it could make recommendations for actions that would affect the DB, but the DB only changed when some external input said the situation had changed. For instance, the planner would read the DS to find out a situation and make some recommendation to move some ships to respond; the DS would be maintaining a cache of some snapshot of the operations DB. This recommendation would go to the operations people (not back to the DS or DB). Operations would decide whether to follow the recommendation. If they did, they would issue an order, which would be executed, which would cause the state of the world to change, which would get reflected in the DB, which eventually would be reflected in the DS.

A DS maintains a cache if the data is bulky, frequently used, or the link is likely to go down. Cache updates can be periodic if either the DB changes regularly (e.g., a target DB where potential targets are identified daily by some upstream planner that works on a daily cycle), the clients run periodically (e.g., more detailed targeting decisions done daily), or the data is assumed to change slowly enough that reasonable decisions can be made with data no more than some age. Cache updates can be forced from the DB if deemed significant (e.g., a bridge destroyed). Cache can be refreshed by a DS query in response to a client request if the client determines it needs current data for some purpose.

There can be multiple DSs fronting the same DB. These would typically be in different CPs, or at least for different functions within the same CP. If there are multiple DSs, their caches may or may not be kept consistent with each other; the issues are approximately the same as for consistency between the DB and a single DS cache.

In the event that a cache update is not received, a client of the DS could hook to the corresponding DS in a peer CP. This would be at slower speed and possibly somewhat different data quality, since the DSs need not have the same view and the peer may have less detailed information. An example would be intelligence information, where the local DS would try to have all detail relevant to the CP's theater, but only a summary of similar information relevant to the theater of a neighboring CP. Accessing a peer DS should be considered an unusual event.

Information locally generated may be stored in the DS cache or separately. This may or may not propagate back to the DB. If not propagated back to the DB, it probably does not go to a peer DS either, although comparable information may be sent as a message.

3.4. Operating in the CP

A CP does all of its processing locally and is assumed to have enough resources for both operations and planning. No tasks are exported to base or peers. If such external processing is required, say for satellite image analysis, it is performed by the base or peer and placed in a DB that is then accessed by a DS in the CP. It does not seem that a CP ever actually schedules anything remotely. This might be because of the command structure, or just to have a looser processing coupling. If the CP needs to have remote work done, it ships the data along with a request, but does not block.

Within a CP, user interfaces and users obviously need to collocate. It is also the case that users doing similar things tend to be collocated as well; hospital staff in hospitals, logistics people together, etc. So similar function will tend to have at least its external parts together.

Each functional area in the CP will have dedicated computing corresponding to dedicated physical resources attached to the unit. In addition, there are pool resources for the CP. There may be 1-2 levels of pool within a CP.

Resources cannot be "stolen" by another CP.

Critical functions are kept up by an "anchor desk" that serves as the root of that functionality. In our terms, an anchor desk is allocated credits to keep its functions going and constitutes a sort of perpetual user. An anchor desk is logically the point of contact for people between functions. There is a distinguished "Survivability Anchor Desk" through which monitoring and modifications of survivability parameters take place.

3.5. Service Configurations

Some services are pinned up by anchor desks.

Shared services may be either pinned open by an anchor desk or created by collaborators for the purpose of the collaboration.

Appendix A-6

A replicated service may be synchronized or may be allowed to drift the same as a DS/DB combination. The clients must specify requirements. This is a place where QoS enters the picture. Since any replica can be accessed by a client, the QoS specification is interesting because it varies with client(s) and replica(s) placement. It is possible to start a new replica close to a client without affecting other client/replica QoS except for the effect of the new replica on the synchronization. This looks to me like a very useful area to explore.

A dedicated service may be started anywhere desired by its client subject to resource ownership, resource availability, executability, and QoS.

3.6. Collaboration

Collaboration takes place in several ways.

Within a given function in a CP, the collaborators are pretty much interchangeable; all air campaign planners (human and program) are more or less the same and work can be partitioned between them. These all access the same DS, so they see the same data.

When collaboration takes place between similar functions between CPs, such interchangeability does not exist, since CPs are in charge of specific theaters or operations (like air), you can't substitute one CP for another in a collaboration without violating these boundaries. Further, peer CPs, even when accessing the same DB, go through a different DS, so they probably do not have exactly the same data sets.

Collaboration between dissimilar functions like weather desk and logistics desk requires those specific functions, although the quality can be allowed to degrade. They are not interchangeable since you can't substitute a weather report for a target list.

4. Example of CP Activity

One of the actions that takes place in a CP is Air Campaign Planning (ACP). ACP is basically the detailed selection of targets to bomb and the generation of the detailed orders to make that happen.

4.1. The Command Decision

ACP starts with a commander (probably a general) evaluating the overall situation and determining the kinds of targets to be attacked in a particular area. For example, communications facilities and airports around Basra. To make this determination, the commander views a large map of the theater overlaid with relevant features. These are the kinds of things you can overlay on a StreetFinder kind of map (e.g., airports, churches, parks), their military equivalent (e.g., airports, factories, communications facilities, intelligence headquarters), and mobile things such as unit positions and weather. There may be similar maps at lower resolution for adjacent areas that are the responsibility of a peer. Generally, organizations stay out of each other's areas in order to prevent confusion and friendly fire casualties. Information about adjacent areas is given because these boundaries are not always perfectly respected and because enemies typically like to exploit the seams between units where the coordination is weaker. In

addition, there will be theater-wide information presented that does not fit to a map. For example, the fact that there is a threat of biological warfare attacks or that certain doctrine such as no attacks on power plants may be in effect. Commanders also may have wire service feeds; in the Data Wall project at Rome Lab, the commander has a CNN feed so he can see what is being said about the operation, since all things have a political side.

4.2. Map Overlays

Each map overlay contains a different kind of information (physical features, threats, weather, logistics availability & position, etc.) and is managed by a separate group of people and stored by them in a separate DB under their control. For example, intelligence analysts identify threats and targets, meteorologists create weather overlays, quartermaster people maintain the logistics overlay, etc. In the JTF architecture, each of these functions would be managed from an "anchor desk". Some of this information is pretty static, such as the base map and many kinds of features, while others such as unit positions and weather may change relatively quickly.

Each overlay DB is created by using (possibly) several more primitive data sources, each of which is in a DB. For example, the threat overlay is produced by intelligence analysts who look at satellite images to find surface to air missile batteries, communications patterns to determine where orders are being sent to/from, ground observations, order of battle information (e.g., an Iraqi division may be known to have a certain number of SAMs, even if you can't see them).

Each kind of underlying data resides in a separate DB closely tied to how it is produced. The data in an underlying DB may be raw or may have been interpreted by some other analyst. For example, satellite images are interpreted by experts who do little else because of the difficulty in interpretation. They would examine raw images and tag and identify (possibly tentatively) items of interest in the images. They would not attempt to ascribe an importance to what they find. There may be feedback from higher levels asking such analysts to look for particular kinds of items (e.g., look for SCUDs), since there is too much imagery for every image to be examined in great detail. Image analysts are supported by software to identify images of interest; for example, successive images of the same area can be diffed to avoid having to redundantly analyze essentially the same image, and it is easy to find edges, reflective surfaces, hot spots, etc., that are likely to represent man-made items. A lot of model based AI tools are used in this to refine the filtering (e.g., "find long objects on trucks near the edge of woods also near a concrete roadway").

Data in the various DBs may be of different ages, as can information within a single DB. Sometimes a history is needed to draw a conclusion. For example, something appearing in a satellite image may be hard to identify, but being able to tell that it moved from one image to the next may give important information about what it is.

An intelligence analyst may access several different kinds of DBs in order to draw a conclusion about a threat. If not all those DBs are available, the analyst will be less certain about the conclusions, but can still function. Similar for other kinds of analysts.

Appendix A-6

All of the analysis activities are supported by software tools of some degree of sophistication. There is a range of processing power needed for these from image enhancers up through weather modeling.

Each of these items that are visible on the overlays can be queried (by mousing) to find its properties as reported by the analysts who prepared them. The general typically will see the conclusions drawn about each kind of item on the overlays and only drills down to the underlying information from which it was deduced as needed. This might be the case if he questions how certain a conclusion is. Management and modeling of uncertainty is a current DARPA research initiative.

Given these overlays, the commander and his staff decide what they want to do. This may involve air attacks against certain kinds of targets in a certain region (airfields and communications near Basra). The commander issues orders that an Air Campaign Plan be made to accomplish this.

4.3. Air Campaign Planning

ACP consists of determining which targets meeting the criteria specified by the commander will be attacked, determining how to attack them, and issuing orders for the attacks. Depending on the number of targets, the campaign may be performed by several air commands. If multiple air commands are involved, the ACP will be collaborative among between the CPs at those air commands. Those CPs will most likely be located at the air commands, which may be airfields or aircraft carriers.

If multiple air commands are involved, the targets are partitioned among them in some way so that planes from multiple commands do not get in each other's way, either at the target or en route. This is done by creating "corridors" in which flights must stay. Corridors have both a spatial and temporal dimension, so that the same space can be flown through at different times. Unless aircraft are to meet somewhere en route, the only coordination between commands are in the choice of targets and corridors during planning and the adherence to corridors during flight.

ACP starts by retrieving a list of targets from the DB that was used in generating the target overlays. More detail may be obtained than was presented to the general or the level of detail may be similar. At this point, the objective is to partition targets among air commands. Once this is done, the CPs stay basically separate except if they want to renegotiate targets or when setting corridor boundaries.

Within a CP, there are many human planners, all of whom do the same set of tasks, only for different targets. Planners query databases to find details about targets, available aircraft and munitions, threats (such as SAMs) than can affect the missions, and information about other features (such as hospitals) to avoid collaterally damaging. The information includes location, importance of its destruction, physical features relevant to attacking it (you need to know how thick a runway is to know what you need to drop on it, and you attack runways lengthwise, not widthwise). Some of this information is generic to the class of target, while other is specific to a particular target.

Individual human planners select targets (individually or in sets) and start assigning resources to attacking them. For each kind of target, there is a list of munitions that can

be used to attack it with varying costs, probabilities of success, and minimization of collateral damage. Different munitions can be carried by kinds of different aircraft. Different kinds of aircraft have different ranges and there is doctrine that tells what kind of environment certain kinds of planes can be flown into ("don't fly this second tier aircraft into an area where you don't have air supremacy"; i.e., only you have any planes left). Planes start at different bases, which tell where they can get to. The planning problem is basically to maximize some objective function consisting of maximizing the probability of the largest number of high value targets damaged or destroyed while minimizing aircraft loss, munitions cost, and collateral damage. The relative weights of these parameters can be changed for different situations, as can the values of targets. Assignment can be done greedily or there can be a lot of hypothetical assignment that can be retracted and resources assigned differently. For example, the optimal assignments to early targets might cause you to run out of the correct munition type for some lower valued target, so you use the next best munition on the high value target in order to have it available for the lower value target. The reason you might get into this situation is that a smart weapon might give a somewhat better way than a bunch of dumb bombs to destroy a key target, but its use there would make it unavailable for use on a lesser target where possible collateral damage to a nearby hospital from a dumb bomb would be unacceptable. There is a lot of work on software planners to aid in this scheduling problem, but for the foreseeable future it appears that human planners will be heavily involved, at least in validating and setting priorities. In the course of this, the planners access DBs of consistency criteria (this plane can't carry that bomb) and policy criteria (under the current rules of engagement, don't damage power plants) that pose additional policy constraints.

Once targets are assigned, orders must be issued to load the correct munitions on the selected planes and give them route instructions. I don't know if this is part of ACP or is delegated to a yet lower level of CP. For our purposes it really doesn't matter. These detailed schedulers will again access parts of the same DBs as the ACP, although they will access only those slices of immediate concern to them. They will also access other DBs for logistics (where exactly are those munitions?), crew information, etc., that are needed for getting a flight scheduled, but that are not needed by ACP.

There might also be a tie into the logistics system to order more munitions and into a maintenance system to schedule preventive maintenance on the planes most heavily used.

Summaries of the orders are issued to AWACS so that they can monitor the mission, including compliance with the corridors.

After missions are completed, their success is evaluated based on information from the same kinds of DBs as were used in the initial planning; the question now is whether the targets are still there and healthy. The cycle then starts over again with the commander deciding which targets to attack; he may choose to attack a different class of targets, even though some of the original ones are intact.

The cycle of ACP and missions is currently (I think) 24 hours, They want to reduce it, but there is some irreducible minimum even if the planning becomes extremely fast, since

missions have to be flown, crews need to rest, and damage needs to be assessed. Thus ACP will always be periodic.

4.4. Processing Assumptions

ACP needs access to a number of DBs that originate external to the ACP function. They will probably be fronted by DSs local to the CP. ACP is performed on a cycle. It is important that an entire plan be consistent to avoid redundant attacks or conflicting operations. This appears more important than using the absolutely most current information. Partly this is because many of the attacks are by human pilots who can compensate or break off an attack where the target has moved. The effect that this "clumpiness" of scheduling has is that it defines the DS cache refresh rate.

It is more important to end an ACP cycle by the time the next flights can take off than to achieve an optimal plan as long as the plan produced is valid. As long as every plan created is acceptable, you can quit at any point you need to when you run out of time and processing power. The way ACP is done takes care of this. In effect, it is OK that the ACP is essentially a batch system, since airplanes must take off in batches also in order to protect each other and coordinate activities. There is no desire to get a plane into the air as soon as it can be given a payload and a target.

Rather than continually update the ACP DS cache, they are attempting to deal with mobile targets by reducing the cycle time so that the snapshot in the cache is less out of date. Although I could not find this, I would guess that the messaging system is used to notify planners that targets have moved or changed in some significant way. Targets no longer valid would then be manually removed from the target list and their resources freed for other missions. This would be similar to what happens when targets and missions are renegotiated between CPs and planners as described previously. If this model is right, that would make the message system a sort of differential DB.

Planners within a CP will grab targets and resources from the DB with little coordination. When they renegotiate and relinquish resources, they will simply put them back in the DB where they will be available for the next planner who accesses them. Planners at this level are physically close together and can communicate these directly, by phone, or by message. Renegotiation between CPs is more significant, since it involves shifting a boundary. Such requests are negotiated by anchor desks at the CPs involved. This is fairly heavy weight and will probably not be done after a lot of detailed planning has happened, since it is likely to invalidate a lot of work already done.

Creating mission orders from target/munitions/aircraft pairings requires more coordination, since the aircraft must physically either fly together or avoid each other. This may be done by a single planner (or small group of interacting planners) taking all flights between a pair of places and coordinating their orders. These flight groups must then be assigned corridors, which requires interaction between the planners for the individual flight groups. There is probably some iteration here, since you don't know what corridor you need until you plan, but you don't know you can get a corridor until you see if it conflicts with someone else.

Appendix A-6

ACP is very intense for a while, and then resources shift to other tasks. Some of these are related to the missions, like monitoring status (loading munitions, air traffic control, etc.), preparing briefing material for pilots and commanders, assessing previous mission success. Others are just other things that go on in the CP that are only tangentially related, like ordering munitions or scheduling. Other activities in the CP are totally unrelated.

A planner probably has a PC class machine that does mostly display of maps, resources lists, messages, orders being produced, etc. It acts as a personal local cache of data being used by that planner. Larger tasks like route finding, scheduling aircraft, checking constraints, etc., are most probably performed on behalf of individual planners on shared machines in the SPARC class or higher. These machines form a pool that can be used for multiple purposes, not all of which are related to ACP. Use of the pool is dictated by the situation. An interesting idea is that as the ACP plan gets better and improvements get smaller, there is lower priority for ACP compared to other activities. Also, there are shifts between the various ACP functions throughout the day. It does no good to have great target selection if that comes at the expense of getting no orders issued; it is far better to have fewer targets selected and actually issue orders to attack them than to select many targets and attack one.

Some DBs including aircraft resources, munitions, and received messages will be local to ACP, while others including maps, targets, weather, doctrine, intelligence, etc., will be remote and will be served by local DS caches. As noted before, several of these caches can be allowed to be out of date and refreshed on a schedule. Each CP has its own caches.

Local plans need not be shared with peer CPs. The partitioning of targets and resources among CPs must be shared, as must the eventual determination of corridors. The DS caches of similar data at peer CPs will in some cases be identical (weather over targets, doctrine, etc.), will sometimes only be similar (features relevant only at one CP will be at a coarser level in peers), ad sometimes will not overlap at all (weather at the peer base). This affects the ability to use cached data from peers should the local DS fail. Note that since DISA has the notion of DB segments (data and DBMS and scripts and schema), it should be possible to download a segment from the DB to replace a failed DS given enough time.

Appendix A-7

Survivability is Utility

David E. Langworthy and David L. Wells
Object Services and Consulting, Inc.
{del,wells}@objs.com

Abstract

The paper explores how Utility Theory (a sub-discipline of microeconomics) can be exploited to define metrics to evaluate the successfulness of survivable systems and that can be used by Survivability Management Systems to plan actions to ensure system survivability. The current lack of such metrics is a serious impediment to progress in the development of survivability techniques.

1. Overview of Survivability Concepts

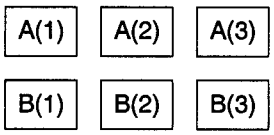
The goal of survivability [1,2] is to provide system-wide integrity well beyond the "islands of integrity" approach of fault tolerance and high availability [3] techniques, which enable reliable data storage and reliable on-line processing respectively. In survivability, the focus moves from hardening individual components to ensuring that every client has access to the services it requires. Survivability takes a variety of proactive and reactive steps in an attempt to keep a system in a state such that it can satisfy the expressed current and future needs of the system's users. To do this, especially using automated tools, requires metrics that can be used to measure the "goodness" of various system configurations that can be reached. Utility theory provides some of these metrics.

While it is true that high availability is extremely useful in providing survivability, availability can be provided without achieving survivability as illustrated by the following short example taken from our proof of concept survivability service.

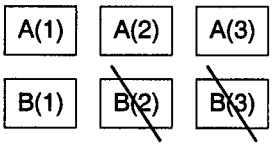
Replicated Service Example: We assume an Object Service Architecture such as CORBA that has been extended to allow a service to be replicated for high availability. In the example, there are only services and hosts. Any differences

This research is sponsored by the Defense Advanced Research Projects Agency and managed by Rome Laboratory under contract F30602-96-C-0330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

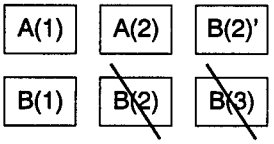
between the hosts or particulars about network topology have been abstracted away. There are two essential services A and B each made highly available with three replicas. Each replica consumes an entire host. In the initial configuration, there are 6 running hosts. At this point each service is highly available and the system as a whole is survivable.



A series of failures or an information warfare attack on B might eliminate both of B's backups leaving A completely intact. A and B's high availability substrate has accomplished its goal. There has been no change in perceived function or performance despite multiple failures. However, it is clear that B has become vulnerable. Since B is essential to the functioning of the system, the system as a whole is equally vulnerable. The system's survivability is compromised. The study of availability does not consider the contribution that a service makes to the overall system.



A survivability architecture manages the system as a whole. The ability of this configuration to deliver functionality into the future has been impaired. If attempts to bring B's replicas back on line fail, the survivability service might try to increase the system wide survivability by sacrificing the availability of A. Since A and B are equally important, a second replica for B is more important to system wide survivability than a third replica for A. Once this realization occurs, the system can be reconfigured to improve its survivability.



After a second replica for B is brought up, this small system is once again balanced. Given the existing resources, the survivability is maximized even though A is now less available than it was before the adjustment. The risk of a failure perceived by an end user is minimized.

2. Measuring Survivability

The usefulness of a survivable system can be judged in several ways:

- how useful is what it is doing now?
- how useful is it likely to be in the future?
- if it breaks, can it be repaired so that it can again do something useful?

There are a number of concepts from utility theory that are helpful in answering these questions. Collectively, they provide us with metrics that can be

used to evaluate the desirability of various system configurations from a survivability perspective. These can be used to allocate resources, plan administrator actions, shed load appropriately as resources dwindle, and warn users about unstable conditions.

3. Applying Utility Theory to Survivability

Utility theory is the study of decision making under risk and uncertainty among large groups of participants with differing goals and preferences [4]. A participant has direct control over the decisions he makes, but these decisions are only indirectly linked to their outcomes. The outcomes depend on the decisions of other participants and random chance. For example, from the perspective of a system administrator attempting to configure a system to survive a collection of faults and threats, a configuration which continues to provide service is clearly preferred to one that does not. However, the administrator cannot directly enforce the preferred outcome; he must choose a relatively small number of administrative actions out of a huge number of possibilities and hope these lead to the desired outcome: a system that operates over time. The actual result will depend on the decisions of other administrators, adversaries, and chance.

The term "utility" is a measure of preference that can be determined and expressed in many different ways. The most common expressions of utility are the supply and demand curves from microeconomics. These graphically represent a supplier's utility for revenue and a consumer's utility for a product. In this case the units are dollars and units sold, both integer values. Utility can also be expressed as an ordinal preference. For example, athletes prefer to come in first rather than second and second over third. A dollar value might be assigned to this preference in professional sports, but this would only be a secondary approximation. Binary utility is relevant to survivable systems. If a user absolutely requires a service, that requirement is either met or failed.

The concept of utility can be used to quantify the goodness of states and actions in a survivable system. System states can be compared using utility measures to determine which is preferred, and as a result, which survival actions should be taken in a attempt to move the system to a better state or avoid worse states.

The utility of a system state or administrative action depends upon the services that are currently running and the future configurations that can be reached. Future configurations need to be considered to differentiate between a rigid configuration that offers good current performance from a flexible configuration that offers slightly lower current performance but is more resilient to faults and is more likely to continue offering good performance. A balance must be reached between present performance and future performance. For example, for most systems the potential configurations a year in the future are not nearly as important as the configurations the system could reach during the next 12 hours.

Utility can have multiple definitions, depending on the overall goals to be achieved. For example, one utility function could value maximizing the work performed, another utility function could value minimizing the likelihood that the level of service provided falls below some threshold, and a third utility function

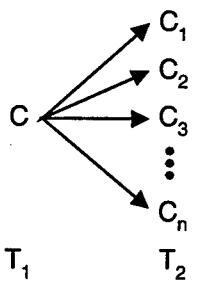
could value minimizing the probability that information is divulged to an opponent. All are equally valid, and depending upon circumstances could in turn be valued to different degrees. This would result in a combined utility function that would be some aggregation of the underlying utility functions.

We now define a model of the system whose utility will be measured and then present several useful utility measures.

System Model: At any point in time, the system is in some *configuration.* A configuration consists of all the resources in the system and the dependencies between them. At the lowest level are physical resources such as computers, networks, sensors and actuators. The dependencies at this layer follow the physical topology of the system. Layered above the physical resource are services. A service can depend upon a physical resource or another service to deliver its functionality. The configuration determines the functionality that users perceive at the moment and the constrains the range of functionality that they will perceive in the future.

The configuration of the system can change as events occur. There are three sorts of events which occur: those initiated by possibly automated system administrators, those initiated by Nature, and those initiated by an adversary. The transitions initiated by administrators are generally beneficial, for example, starting a new replica or shutting down a system for scheduled maintenance. A detrimental administrator-initiated action such as shutting down the last operational replica in an attempt to repair a different one is relegated to Nature in this analysis. We blame Nature for the entire spectrum of faults that typically plague a computer installation including power failure, disk crashes, programmer errors and the like. The only time Nature acts in our favor is when it stops, for example, when it stops raining and our microwave link works again. An adversary is capable of coordinated malicious actions and deceptions. These actions include launching viruses, physical destruction of sites, and security violations which compromise sensitive data.

In general, an event can occur at any time. Each event has a probability of occurring within some time interval and events can be correlated. The probability of a configuration existing at a certain time in the future is computed based on the current configuration and the probabilities of the transitions required to reach the new configuration. The example below shows one configuration, C, at time T_1 branching into n possible configurations at T_2.



This computation is necessarily complex because multiple paths of transitions lead to the same configuration. To simplify the problem, our current analysis divides time into discrete, fixed size intervals, and we concern ourselves only with

configurations at time interval boundaries. This allows multiple events to occur during a time interval. Under this simplifying assumption, the fanout at each time step is still very large, so exactly calculating the range of possible configurations even a small number of steps into the future is infeasible. Further, calculating the probability that an adversarial action occurs requires game theoretic constructs. Also, except in trivial cases, transition probabilities can be at best estimated. In future work, we plan to develop a means to project a configuration along "interesting" paths that provide a representative sample of the space of possible configurations at some future time, to model continuous time and variable size time intervals, and to analyze the effect of imprecise estimates of transition probabilities.

In any configuration, every client of a resource (at any level of abstraction) receives a benefit from receiving the service the resource provides. This is expressed as a *benefit function, B,* that maps a description of the service being provided to a value received. The service to be received can be described in many ways, including using quality of service (QoS) concepts such as timeliness, precision, and accuracy of the results to be provided. This paper does not address the particular form of a service specification.

The benefit a client receives from a service is accrued only if the service completes its task; i.e., an instantaneous, ephemeral connection to a service provides no value. Thus, every benefit function must include a *duration* over which the service must be provided in order to attain the specified benefit. In principle, this could be a number of invocations or a time period. In our current analysis we restrict the duration to the fixed size discrete time intervals defined above; a client receives the benefit only if the service is still being provided at the end of the interval. Again, we hope to generalize this to continuous time and mixed interval sizes in future work.

Utility: In general, utility is a measure of the desirability an outcome. In this case, we define the *utility of a configuration, U(c),* to be the aggregation across all clients in a configuration of the value of the services they receive. Recall that utility can have multiple definitions, depending on the overall goals to be achieved. All are equally valid, and depending upon circumstances, could in turn be valued to different degrees. This would result in a combined utility function that would be some aggregation of the underlying utility functions. Because there can be multiple utility functions, we differentiate between them using subscripts when necessary; e.g., $U_{work}(c)$. Different utility functions are created by defining different aggregation functions. Two of these utility measures are discussed in more detail below. For now, it is sufficient to accept that for any given configuration, it is possible to compute the values of one or more utility functions.

Expected Utility: As with the benefit provided to a client by a service, a configuration provides a given utility only for tasks it completes. In our current model, this is determined at the end of a time interval. Since a system that begins a time interval in some configuration c may end it in some other configuration that provides a possibly different utility (based on the services that it the new configuration provides), a more useful measure of utility is the *expected utility of a configuration c, EU(c).* EU(c) measures the benefit of a collection of potential configurations, C, that can be reached from c in one time interval. It is the

probability weighted sum of the utilities of each individual configuration that can be reached. The probability function, $P(c_i)$, is the probability of c_i being instantiated out of all the configurations in the set. Of course, the probabilities must sum to 1. The set, C, is subscripted with time, so we are measuring the probability that a configuration is instantiated at some particular time in the future. In the notation, a lower case 'c' indicates one particular configuration and an upper case 'C' indicates a probability distribution over a set of configurations.

Expected Utility = EU(c) = $EU(C_T) = \sum_{c \in C_T} P(c) \times U(c)$

The above computation assumes that negative (natural and adversary) events can occur at any time during the time interval, but that helpful (administrative) events take place only at interval boundaries. In other words, it is impossible to fix a problem during an interval. This is realistic, since administrative actions take some time to complete, whereas natural and adversarial actions, even if they take time, are generally not noticed until their effect on the configuration is felt.

Because there can be more than one base utility function (e.g., $U_{work}(c)$), there will be more than one expected utility function (e.g., $EU_{work}(c)$).

Net Utility: Expected utility allows us to compute the benefit that can be expected to be obtained from a configuration even after considering the near term negative events that can cause the configuration to degrade. However, we now need another kind of utility measure to allow us to consider longer term changes to the system and to incorporate the ability to perform beneficial administrative transformations. We call this *net utility, NU(c).* Net utility measures the fact that the long term desirability of a configuration depends upon the services that are currently running and the future configurations that can be reached. Net utility is thus a sum of future expected utilities. In general, not all time periods are of equal importance; as noted previously, the near term behavior of a system is usually valued more highly than behavior far into the future. To handle this, we introduce a discount function, D(T), which maps from time to an appropriate weighting factor. The discount function is related to net present value in finance.

The following equation calculates the Net Utility, NU(c), of a configuration based on the discount function and Expected Utility, EU(C), defined above.

Net Utility = $NU(c) = \sum_{t>now} D(t) \times EU(C_t)$

In natural language, the equation reads "the net utility of a configuration, c, is the sum of the expected utility of the potential configurations at each time step into the future discounted by the appropriate factor."

Both NU(c) and D(t) are specialized to the particular kind of base utility function; e.g., $NU_{work}(c)$.

The use of a discount factor has an additional benefit, since it allows us to discount far future states for computational as well as policy reasons. This has a practical advantage, since as noted previously, when one projects the configuration space further into the future, the computations rapidly become more expensive (due to state explosion) and the results rapidly become less precise (due to imprecise estimates of event probabilities). The benevolent myopia introduced by the discount factor allows us to ignore incomputable or dubious future states.

Appendix A-7

Alternative Utility Metrics: The meaning and power of the utility functions defined above vary greatly depending on the precise definition of the base utility function U(c). As noted above, the base utility function measures what is valued most highly. We now describe two possible metrics.

The first survivability metric we developed, *Utility of Value*, was based on a measure for aggregate performance. This work developed from a market based, distributed resource allocation prototype. The goal of the market was to maximize the value of all the services provided by the system. End users or administrators would assign values to services. The resources, both hardware and software, would compete to offer the best service at the lowest cost. The resources' goal was to accumulate profits which would be gathered by the owners of the resources and allocated to end users and administrators closing the loop.

Utility of Value is sufficient to solve the problem presented in the example at the beginning of the paper. If users value a service highly, it will replicate itself to assure that it is highly available. If resources are removed from the system, the prices will rise and only the more valued services will obtain resources. Likewise if resources are added, prices will fall and lower priority services will run. It implements a simple microeconomic model that tends toward Pareto Optimality, a local optimality criterion.

If the Net Utility of Value is maximized, then future performance of the system will be maximized. There are many possible definitions of survivability, but a relatively straightforward one is that the system continues to offer good performance into the future. Value ranges over the integers depending on how well the system is performing. More value is always better and less is always worse.

Our second metric, *Utility of Operation*, is based on a binary measure depending on whether the system meets some minimal level of operation over a given interval. This gives rise to a very different notion of survivability. Using this measure, EU(C) is itself a probability: the probability that the system is operational. Maximizing the Net Utility of Operation minimizes the possibility of some catastrophic failure in the future, possibly at the cost of optimal average case performance. This is arguably a better survivability metric than the Net Utility of Value, since the purpose of survivability is to avoid catastrophic failures. The two could be used in conjunction so that after a minimal level of service is guaranteed, performance is optimized for the normal case.

Note that both Utility of Value and Utility of Operation differ significantly from resource utilization in that they measure the perceived benefit of the system, not how hard the system is working.

4. Examples

We now present four examples to illustrate how Utility of Value and Utility of Operation are computed and the differences between them.

Example 1: The example with which we began the paper is a simple illustration of Utility of Value. Returning to the example above, three configurations are illustrated. We will quickly review the example with a

condensed notation. There are two services, A and B, and six hosts, 1 - 6. In the initial configuration, C1, each service has three replicas: C1 = A{1, 2, 3}; B{4, 5, 6}. After the failure, B loses two replicas, C2 = A{1, 2, 3}; B{4}. The third configuration, C3, is the result of a possibly automatic administrative action which trades a second backup from A to provide a single backup for B, C3 = A{1, 2}; B{4, 3}. This last transition is voluntary. The administrator or survivability service would take what ever action seemed best.

The system is very simple and in this analysis we will look only one time step into the future and consider two possible transitions, failure and startup. Each service, A and B, is given a theoretical value of 1000 and requires an entire host for a primary or a backup. This value is only reached if the service runs without error through the period. There is a 10% probability of failure of each host during a period, so the probability of success of a service with n replicas is $1-0.1^n$.

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
AB	2000	.9980	1996	.8991	1798	.9801	1960
$A\overline{B}$	1000	.0009	9	.0999	100	.0099	10
$\overline{A}B$	1000	.0009	9	.0009	1	.0099	10
$\overline{AB}$	0	.0000	0	.0001	0	.0001	0
			1998		1899		1980

This table calculates the expected utility for each configuration in the example. The first column indicates the state of the system. A bar over the service label indicates the service is not operational at the end of the period. For example, $A\overline{B}$ indicates that A is running but B has failed. The second column is the value of the configuration. Here the aggregation function is simple addition, so if both A and B are operational the value of the configuration is 2000. The third and forth columns show the calculation of the expected utility of C1, EU(C1), which is the total shown on the last row of the table. The fifth and sixth carry out the calculation for C2 as the seventh and eighth do for C3.

In C1 everything is running fine. Out of a possible value of 2000 the expected utility is 1998, almost perfect. After the failures, the expected utility drops to 1899 because of the uncertainty that B will complete. C3 reflects the administrative action of taking a replica from A and giving it to B. This increases the expected utility to 1980, a dramatic improvement considering that no resources were added.

Example 2: The next example calculates the binary Utility of Operation using the same probabilities and configurations. The difference is in the utility of each configuration. Since both A and B are required services, each is given a value of 1. The aggregation function is a logical AND, so only the configuration with both operational is given a value of 1; all others are 0.

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
AB	1	.9980	.9980	.8991	.8991	.9801	.9801
$A\overline{B}$	0	.0009	0	.0999	0	.0099	0
$\overline{A}B$	0	.0009	0	.0009	0	.0099	0
$\overline{AB}$	0	.0000	0	.0001	0	.0001	0
			.9980		.8991		.9801

The utilities are calculated as before. When calculating the expected Utility of Operation the result is no longer binary. It ranges from 0 to 1 and is the

probability that the configuration will maintain a minimal level of operation through the period. In C1 everything is running properly and the expected Utility of Operation is 0.998, nearly perfect. After the failures, the expected utility drops to .8991, again largely due to the risk that B will not complete. Again, the C3 reflects the administrative action of shutting down an A replica and giving it to B which significantly improves expected utility.

Example 3: So far, the two service utility metrics produce the same desired configurations. However, if multiple levels of QoS are introduced, it becomes apparent that the two metrics differ substantially. This next example introduces QoS. The service A now has two levels of operation, high and low. The high level offers a value of 2000 and requires 3 hosts to run. The low level is required for a minimal level of operation and offers a value of 1000 but requires only 1 host to run. If the high level of service cannot be maintained, it automatically drops to the low level of service. In the example A starts out at the high level of QoS. If A loses a host, it drops to the low level of QoS with one replica. The probability that A completes the period at the high level is the probability that all three hosts complete. The probability that A completes the period at the low level is the probability that any single host completes minus the probability that A completes at the high level. There are now 6 possible outcomes. B is still worth 1000, so if A completes at the high level along with B the value is 3000.

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
A_hB	3000	.7283	2185	.6561	1968	.0000	0
A_lB	2000	.2697	539	.2430	486	.9801	1960
$\overline{AB}$	1000	.0010	1	.0729	73	.0099	10
$A_h\overline{B}$	2000	.0007	14	.0270	54	.0000	0
$A_l\overline{B}$	1000	.0003	0	.0009	1	.0099	10
$\overline{AB}$	0	.0000	0	.0001	0	.0001	0
			2739		2582		1980

In the initial configuration all hosts are operational and the expected Utility of Value is nearly optimal at 2739. After the failures, the expected value drops by about 150 reflecting B's instability. C3 evaluates the administrative action of removing a host from A to increase B's stability. In this case, the action does not appear to be desirable and would not be taken. The reason is that removing a host from A would cause it to drop from a high level of QoS to a low level of QoS at a cost of nearly 1000.

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
A_hB	1	.7283	.7283	.6561	.6561	.0000	0
A_lB	1	.2697	.2697	.2430	.2430	.9801	.9801
$\overline{AB}$	0	.0010	0	.0729	0	.0099	0
$A_h\overline{B}$	0	.0007	0	.0270	0	.0000	0
$A_l\overline{B}$	0	.0003	0	.0009	0	.0099	0
$\overline{AB}$	0	.0000	0	.0001	0	.0001	0
			.9980		.8991		.9801

Example 4: The goal of the Utility of Value metric is to maximize perceived performance and maintaining A at a high level of QoS is consistent with this goal. However, the survivability of the system is sacrificed by this choice as the next example using Utility of Operation shows.

In the initial state all hosts are operational and A is operating at the high level. After the failures, B is reduced to one replica and the expected Utility of Operation drops to .8991. A is still operating at the high level, but this is not reflected in the binary operational metric. Step 3 reflects the administrative action of taking a host from A. This causes A to drop from the high level to the low level and increases the stability of B. As a result the expected operational utility increases to .9801.

This example illustrates the difference between the integer Utility of Value and the binary Utility of Operation. Utility of Value optimizes for performance and Utility of Operation optimizes for stability. Which objective is preferable depends on the situation. We would actually like to achieve both with some sort of hybrid measure. For example, if two configurations are within an epsilon of expected operational utility, then choose the configuration with the highest value. The size of the epsilon would be controlled by an administrator and could vary over time. For example in peace time, performance is preferable; however, before an engagement the value could be tightened down to reflect an increased need for stability.

5. Conclusions

We have presented a simple computational system model over which survivability metrics can be computed, defined a series of metrics based on utility theory to measure the immediate and long-term desirability of system configurations, and presented two specific objective functions that value different system properties (work and resilience). These were illustrated by examples. We identified several extensions that should be investigated in the future to provide greater fidelity between the models and reality.

6. References

1. "Survivability in Object Services Architectures - 1998 Annual Report", David Wells, Object Services and Consulting, Inc., http://www.objs.com/Survivability.htm, 1998.
2. "DARPA/ITO Information Survivability Website", Defense Advanced Research Projects Agency - Information Technology Office, http://www.darpa.mil/ito/research/is, 1998.
3. "Lazy Replication: Exploiting the Semantics of Distributed Services," R. Ladin, B. Liskov, L. Shrira, Proceedings of the Ninth Annual ACM Symposium on Principals of Distributed Computing, Quebec, 1990.
4. "Game Theory in the Social Sciences: Concepts and Solutions," Martin Shubik, MIT Press, Cambridge Massachusetts, 1982.

Appendix A-8

Survivability in Object Services Architectures

Estimating Service Failure

David L. Wells
David E. Langworthy

Object Services and Consulting, Inc.

In the process of creating and maintaining survivable configurations, the Survivability Service needs to predict the likelihood that, within some time interval, a service will be damaged to an extent that it cannot provide the required level of service. This paper discusses a basic model of how services are provided by resources, how threats against those services are modeled, and how the probability of service failure is computed from the threat model.

---

1. Introduction

While creating and maintaining survivable configurations, the Survivability Service needs to predict the likelihood that, within some time interval, a service will be damaged to an extent that it cannot provide the required level of service.

This report presents a general model of how services are provided by resources, how threats against those services are modeled, and how the probability of service failure is computed from the threat model. The general model allows substantial flexibility in the details of how resources and threats are modeled; this constitutes and important parameterization of the Survivability Service.

In order for the Survivability Service to function, at least one specialization of the general model must be defined; the model specialization currently used by the Survivability Service is presented. Of course, in order for the Survivability Service to actually function, there must be a concrete realization describing the properties of the particular software system being monitored, specified as defined by the model specialization being used. Such a concrete realization is outside the scope of this report.

In general, the failure computations can be computationally intensive. We present a number of optimizations to reduce the effort required to estimate failure probabilities.

The report conclude with issues related to the general model and our particular specialization.

2. General Model

Estimation of service failure requires a model of resources available within the system, a model of threats to those resources, and calculating techniques to apply the threats to the resources. This

section defines the most generalized form of each of these that must be adhered to by any specialization.

The generalized model must be able to express:

- independent threats to individual resources,
- attacks that affect multiple resources,
- multiple attacks against the same resource,
- correlated attacks on groups of resources,
- variable probabilities of resources being affected by attacks,
- ongoing attacks,
- variable amounts of damage by attacks,
- simple resources that do not depend on other resources,
- complex resources that depend on other resources, and
- multiple levels of resource quality of service.

Not all of these need be supported by a particular specialization.

2.1. General Resource Model

Resources. Every service is provided by a *resource*. A resource may be either a *base resource* or a *complex resource*. Complex resources use the services provided by other (base or complex) resources; base resources do not. A base resource is affected only by its own failure, while complex resources are affected by the failure of resources upon which they depend. It is typical that services specify only the services upon which they directly rely. However, nested dependencies exist whenever a resource relies upon the services of a complex resource which will itself have dependencies. In this case, a failure of a lower level resource may cause the failure of a higher level service. We define the *resource closure of a service* in the obvious way. The resource closure may be represented as a directed graph of resource dependencies.

Any *cut* through the resource closure graph of a service represents a set of resources, either base or complex, on which the service relies. Such a set is called a *resource group* of the service.

A resource group R is denoted as:

$R = \{r_1, .., r_n\}$
where:

r_i is an individual resource (base or complex) in R

There are potentially many (equivalent) resource groups for a given service, depending on where the cut is made. Only one of these contains exclusively base resources.

Consider a simple planning service example. The planning service directly relies upon five resources: its own code, the computer the planning service currently runs on, a map service, a weather service, a network connection to the map service, and a network connection to the weather service. The map and weather services are themselves complex services, while the planning service's code, its computer, and the network connections can be modeled as base resources. The map and weather services, each rely on a host and some software. The planning service does not directly care (or even know) about the resources used by the map and weather services as long as those services continue to perform. However, since a map host failure will cause the map service to fail, which will in turn cause the planning service to fail, these base resources are included in the resource closure of planning service and consequently are of concern to the Survivability Service.

Resource States: Damage causes the *state* of a resource to change (unless it was already failed). A specialization of the resource model must define the potential states of each resource. More states increases the realism of the model at the cost of increased storage and computational complexity. Natural choices for resource states are:

- {Fail, Run} if resources are considered to be either functioning perfectly or not at all (or at least insufficiently well to be useful),
- percentage degradations (e.g., lose 20% of capacity) to describe diminished capability to provide QoS for resources whose level of service is easily measured (e.g., CPU cycles), or
- QoS states (e.g., High, Low, Failed) for services whose QoS level is not easily measured quantitatively, or where there is a desire to reduce the number of QoS levels to a more manageable number as is done in the QuO system.

Just as individual resources will be in some state, so will resource groups. The *state space of a resource group* is the cross product of the potential states of the individual resources in the resource group. A *state* of a resource group is one particular element in this state space. The state space of a resource group grows rapidly with group size and number of states for the individual resources. For this reason, it is important that there be ways to prune portions of the space as irrelevant for common cases. These are discussed below.

The state of a resource group is important because we generally care less about the state of individual resources than we do about the state of configurations of resources. There are three reasons for this:

- providing a complex service requires many resources of different kinds,
- not all resources in a resource group are necessarily required, and
- there is often substantial freedom in the choice of configuration; if one doesn't work, another might.

A resource group will be in exactly one state at any given time. Since a service is itself a resource, it

will be in some state as well. There is a mapping from resource group states to service states that indicates the state(s) possible for the service given the states of the resources. Note that this is not strictly a function, since a service might choose to operate in a lower state than is possible given its resources (for example because its client does not currently demand a higher level of service). In general, the service will be attempting to operate in some state, so it will be possible to tell, from the state of its resource group, whether it will be successful.

Probability Distribution of State Spaces: It is common to need to estimate the probability that a service will be able to function in a particular state given its current (or planned future) resource group. This means that it is useful to describe the potential of a resource group for being in any of its possible states. The *probability distribution of the state space of a resource group* is an assignment of a probability to each potential state. Naturally, the probabilities must be in [0.0 .. 1.0] and must sum to 1.0.

2.2. General Threat Model

Threats: All resources are subject to threats. A *threat* is the potential for damage to some collection of resources. These resources are said to be *vulnerable* to that threat and are called that threat's *vulnerable resource group*. Vulnerable resources can be damaged if the threat manifests itself as an *attack*. An attack can cause *damage* to zero or more of the vulnerable resources. Sometimes this damage is correlated (resources are likely to be affected together in the same way) and sometimes it is not. When a resource is damaged, it moves to one of its lower potential states depending on the nature of the threat. Damage to resources changes the state of the resource group.

A threat T is defined as:

T = (R, D(R))
where:

- R is the vulnerable resource group
- D(R) is a *damage effect matrix* specifying the probability of transitioning, within some time interval, from any resource group state to any other resource group state because of damage from an attack

Note that D(R) defines the probabilities of transition within some time interval. D(R) can be huge, so for pragmatic purposes, it is important to define a threat model in which relevant parts of D(R) can be computed from a smaller specification. This is discussed below.

Restricting Threats to Resource Groups: Ultimately, we care about the effect of threats on services' resource groups; i.e., being able to compute the probability distribution of the state space for arbitrary resource groups given known threats. It will frequently be the case that the vulnerable resource group for a threat contains resources that are not part of the resource group for a service under investigation and *vice versa*. While an attack based on the threat may cause resources outside the service resource group to fail, we will generally be unconcerned with this. For this reason, it is useful to be able to *restrict* a threat to only those elements in the service resource group. A *restricted threat* is one whose definition has been modified by subsetting the vulnerable resource group and making modifications

to D(R) to limit transitions to the subset with the appropriate transition probabilities.

If a resource group V is vulnerable to a threat T, then any subset of V is also vulnerable to T. The implication of this is that the intersection of the vulnerable resource group of a threat and the service resource group of a service is also a vulnerable resource group for the threat. This means that for any threat T with resource group R_T and service S with resource group R_T, it is possible to compute the definition of a restricted threat T' whose vulnerable resource set is this intersection and whose stochastic state transition matrix applies only to the intersection. Resources in the service resource group that are not vulnerable to the threat cannot be affected, but must be considered when defining the damage matrix.

A restriction T' of a threat T to a service S is defined as:

- T' = (union(intersect(R_T, R_S), R_S), D(union(intersect(R_T, R_S), R_S))

The computation of D(union(intersect(R_T, R_S), R_S) is dependent upon the form of D(R_T) and D(R_S). The way to compute this for our specialization of the threat model is discussed below .

Threats to Complex Resources: In principle, it would be possible to define threats that applied to entire services. However, this is unsatisfactory for a variety of reasons:

- It is far harder to make realistic estimates of the vulnerability of complex things (services) than simple things (base resources). Threats to base resources are generally simpler and more uniform because of the uniformity of base resources. For example, the threat of hardware failure is the same for all identical SPARC stations and all copies of the same code will have the same programming errors. As a result, all functionally similar resources are subject to the same threats. Complex resources, on the other hand, are constructed from simpler resources and they may be constructed in a variety of ways. Two functionally equivalent map servers may utilize very different kinds of base resources and thus be subject the quite different threats.

- Many kinds of threats to base resources are already known. For example, analyses of hardware failure rates are common, insurance companies make their living by accurately estimating the likelihood of physical disasters such as fires and floods, and the SEI and others are working on ways to measure software quality. The same is not true for complex services that rely on many resources.

- The number of individual services for which threats would have to be individually specified is daunting,

- Modeling threats against directly against complex services ignores the fact that services can be reconfigured to use different resources.

- Modeling threats against directly against complex services ignores correlation between failures of different services. This is particularly critical when considering replicas, where correlated failures may make a system far less robust than it might appear on the surface, or when two services rely on the use of common base resources.

For this reason, we choose to directly model threats against base resources and aggregate them to determine the threats against more complex resources. Threats to base resources are identified and defined by a human and stored in a database accessible to the Survivability Service. Threats to complex resources are computed from these. Note that if a service is a black box, it can be modeled as a base resource, but then the Survivability Service will have little latitude in helping it survive.

2.3. Computing the Effect of Threats on Services

2.3.1 Overview of the Calculations

Computing the effect of threats on a service proceeds as follows:

1. The resources used by the service are determined. This will typically require recursively following sequences of single level resource dependencies through intermediate services down to base resources. Base resources must be reached, since it is only for base resources that threats are stored in the concrete threat model. The resource dependencies may be represented as either a directed graph whose intermediate nodes are intermediate level services, or flattened into a state space of base resources.

2. The states of the resource group that will allow the service to function as desired are determined. This information is used to derive a function that maps the probability distribution of the resource group's state space to a probability that the service continues to perform.

3. The damage effect matrices of the threats (as restricted to the resource group) are applied to the resource group to determine the probability distribution of the resource group's state space.

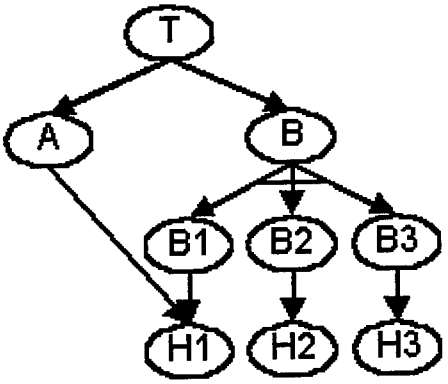
4. The probability of the service being able to perform as desired is computed from the probability distribution of the service resource group's state space using the function derived in step #2.

2.3.2. Computing with a Flattened State Space

The most general form of the computation flattens the resource group to base resources. For even moderate sized resource groups, this computation is intractable, so a number of optimizations are used to manage the computational complexity. However, because it represents the most conceptually clear form of the calculation, it is presented first, followed by the various optimizations.

The resource group is represented by a state space formed as the cross product of all possible states of all base resources in the resource closure. The resource dependency graph can be used to determine which states will allow the service to perform and which will not. States of a resource group that allow a service to function can be framed as a Boolean expression over the resources in the resource group. For each complex resource in the expression, replace it by the Boolean expression describing states of its resource group needed to function. Repeat this until all complex resources are eliminated. The resultant expression can be simplified if desired.

Consider the example below.



In the example we have a high level resource T. T depends on a service A which runs on host H1. In addition, T depends on a replica group B. B depends on three replicas B1, B2, and B3. In the diagram the bar across the links indicates an "or" dependency rather than the standard "and" dependency. B1, B2, and B3 depend on H1, H2, and H3 respectively. Service A runs application code A' and replicas B1, B2, and B3 all run application code B'. The replicas of B are coordinated by code C'. For simplicity, assume that a resource (host or code) either performs or fails.

T performs if A and B both perform. A performs if A' and H1 both perform. B performs if C' and any of B1, B2, or B3 perform. Bi performs if B' performs and Hi performs. Thus, a state in the resource group state space will allow T to perform if it satisfies:

- `(A' and H1) and (C' and B' and (H1 or H2 or H3))`

Naturally, alternate forms of this expression are possible.

The probability distribution of the state space is computed iteratively by applying the damage matrices of the restricted threats. At each iteration, we will have the probability distribution for the state space given threats that have been considered so far. The initial state (actual or believed) is determined directly from the monitors (sensors).

Once the probability distribution of the resource group state space has been computed, the probability that the service will function is a simple matter of summing the probabilities of being in each of the states that satisfy the function computed above.

2.3.3. Generic Optimizations

The procedure outlined above is computationally intensive and creates large closures. Fortunately, in many configurations, it is possible to simplify this process substantially.

Optimizations of the following kinds are possible, depending on the particular model specialization chosen:

- reducing the size of the resource group state spaces, which will in turn shrink the size of the matrices for state space probability distributions and damage effects,
- specifying threats in such a way that it is possible to compute (rather than specify) the damage

effect matrices in most cases; to do this, we differentiate between threats to base resources and threats to complex resources,

- pruning techniques to be able to coalesce equivalent states or to avoid computations altogether when their effect is insignificant,
- recursive techniques to hide inherited resource dependencies when considering complex services with many levels of dependencies to reduce state space explosion while not losing the effect of threats against possibly shared lower level resources.

The optimizations mentioned above make use of properties of the kinds of configurations maintained by the Survivability Service. Our *survivable object abstraction* (see Composition Model of OSA and Evolution Model of OSA) places certain restrictions on the form of resource dependency graphs. At any single level of abstraction, there are three basic resource dependency structures:

- all of a collection of potentially diverse resources are required,
- at least one of a collection of functionally equivalent replicas are required, and
- some quorum q1 of functionally equivalent replicas are required.

These techniques are applicable to many specializations, but for clarity, we will explain them in the context of our own specialization.

3. A Specialization of the General Model

The general resource model, threat model, and calculations presented above are very powerful. The general model leaves free the choice of possible resource states and the algebra for the damage effect matrix, since this degree of freedom is necessary in order to support different survivability and QoS policies. A specialization must define these. Also to be defined is a concrete representation for resources and threats.

While specializing the general model, it is desirable to keep in mind that the general model is extremely space and computation intensive, so it is important that the specialization be amenable to optimizations that make specification, storage, and computation tractable. This section presents such a model. It satisfies most of the desired properties (reiterated below) provided by the general model and is computationally much more efficient. Desired properties satisfied by the specialization are denoted by a "*".

- independent threats to individual resources (*),
- attacks that affect multiple resources (*),
- multiple attacks against the same resource (*),
- correlated attacks on groups of resources (*),
- variable probabilities of resources being affected by attacks (*),

- ongoing attacks (*),
- variable amounts of damage by attacks,
- simple resources that do not depend on other resources (*),
- complex resources that depend on other resources (*), and
- multiple levels of resource quality of service.

3.1. Resource States and Damage Effects

Restricting the resource states that will be modeled is key to making the computation tractable, since if there are m allowable states, a resource group with n elements will have m^n states. If an attack can cause any combination of resources to transition to any lower resource states, then for m states and n resources, the damage effect matrix must contain $(m*(m+1)/2)^n$ entries. [Note: proof by induction on m.]

We consider only two possible states for resources: {Failed, Running}. The state space of a resource group is thus $(F, R)^*$. This also means that the damage effects are limited to {Fail, NotFail}.

3.2. Threats to Base Resources

As noted above, there are several differences between threats to base resources and threats to complex resources that cause us to only directly specify threats to base resources. These same differences allow us to model threats to base resources in a simpler, more compact fashion than is required for threats to arbitrary (potentially complex) resources by the general model. In particular, there is a much simpler, more compact way to specify the potential effect of a threat that eliminates the need to specify a complete damage effect matrix for threats to base resources.

3.2.1 Threat Specifications

Recall that in the general model, a threat T is defined as:

T = (R, D(R))
where:

- R is the vulnerable resource group
- D(R) is a *damage effect matrix* specifying the probability of transitioning from any resource group state to any other resource group state because of damage from an attack

As seen, D(R) can quickly become huge, so there is an incentive to create a more compact form. Even with the only resource states being {Running, Failed} and the only transitions being {Fail NotFail}, the size of of the damage effect matrix for an n element resource group is 3^n. The matrix can be further reduced in size by observing that Fail(X)=Failed and NotFail(X)=X. Thus, it is sufficient to give the probabilities of failure for all possible combinations of resources in the group. This is 2^n entries (probabilities) for a resource group with n elements.

However, this is still large. We can do better by observing that a resource is only damaged if a threat materializes as an *attack* and by then assuming that in the event of an attack, all base resources have equal probability of failure. A specification of this type looks like:

$T_{simple\text{-}base} = (R, P(A), P(Fail(r)|A))$
where:

- R is the vulnerable resource group
- P(A) is the probability of an attack materializing in the time period as the result of the threat
- P(Fail(r)|A) is the probability that each resource in R fails if an attack occurs

From this, the stochastic state transition matrix can be computed easily (see below).

This model is realistic for many kinds of threats. The following examples show how different situations can be represented in this model of base threats.

Example #1 - Independent Failures: A and B are hosts in different mobile platforms, The threat is that the platform (and hence the computer) will be destroyed; there is a 0.01 probability of that happening during our interval. The platforms are independent, and equally likely to be destroyed.

$T_{destroyed} = (\{A, B\}, 1.0, 0.01)$

Example #2 - Attacks Against Multiple Resources: C and D are hosts that rely on the same power supply. The threat is that the power supply fails; there is a 0.01 probability that is happens during our interval. If this happens, both A and B will fail; if it does not happen, neither will fail as a result of that threat.

$T_{power} = (\{C, D\}, 0.01, 1.0)$

Example #3 - Correlated Attacks: E and F are programs that are susceptible to a viral attack. There is a 0.04 probability that such an attack will be launched during our interval, and that if it is, there is a 0.25 probability that each of E and F crash.

$T_{virus} = (\{E, F\}, 0.04, 0.25)$

3.2.2. Computing the Damage Effect Matrix

The restriction of the damage effect matrix for a threat to a group of base resources is computed as follows. Assume the restricted vulnerable set R has n members. In our model, failures are independent if an attack occurs, so the number of failed resources, m, in the event of an attack is binomially distributed; i.e., $p(n, m)= P(Fail(r)|A)^m * (1-P(Fail(r)|A))^{n-m}$. In addition, no resources fail if no attack occurs. So:

- P(transition to a state with m0 failed resources) = $P(A) * P(Fail(r)|A)^m * (1-P(Fail(r)|A))^{n-m}$
- P(no failures) = $(1 - P(A)) + (P(A)*(1-P(Fail(r)|A))^n)$

Resources outside the vulnerable resource group cannot be affected by the threat, so the probability of transition to states in which they Fail is zero.

Example #4: T_{virus} ({E, F, X}, 0.04, 0.25) applies to a service with resource group {E, F, Y}. The restriction of T_{virus} is:

$T_{virus}' = (\{E, F, Y\}, 0.04,$

```
[(RRR, 0.9825)      // 1 - 0.04 + 0.04*(1-0.25)^2
 (RRF, 0.0000)      // Y cannot fail
 (RFR, 0.0075)      // 0.04 * 0.25^1 * (1-0.25)^1
 (RFF, 0.0000)      // Y cannot fail
 (FRR, 0.0075)      // 0.04 * 0.25^1 * (1-0.25)^1
 (FRF, 0.0000)      // Y cannot fail
 (FFR, 0.0025)      // 0.04 * 0.25^2 * (1-0.25)^0
 (FFF, 0.0000)])    // Y cannot fail
```

3.2.3. Expressiveness of the Specialization

Examples #1-3 represent, respectively, independent failures, attacks against multiple resources, and correlated attacks. Consider the following table, which shows the probability that particular configurations of AB, CD, and EF are functioning at the end of our interval. An R indicates the resource is running, an F indicates it has failed.

AB	P		CD	P		EF	P
RR	0.9801		RR	0.9900		RR	0.9825
RF	0.0099		RF	0.0000		RF	0.0075
FR	0.0099		FR	0.0000		FR	0.0075
FF	0.0001		FF	0.0100		FF	0.0025

Since we care about which subset(s) of the resource group are likely to end up in, this distinction matters. Since the important subsets are often either the entire resource group (i.e., [RR]) or those with at least one surviving member (i.e., not [FF]), the need to consider correlation is clear. The case where a resource is shared among the resource groups of multiple services is similar.

This specialization accurately captures two critical survivability notions (assuming that failure probabilities are similar):

- when all resources in a resource group must survive, it is best to choose resources whose failures are correlated, since this reduces the number of ways in which resources can be compromised
- when only some (e.g., replicated) resources in a resource group must survive, it is best to choose those whose threats have as little correlation as possible, since this prevents a small number of attacks from affecting many resources.

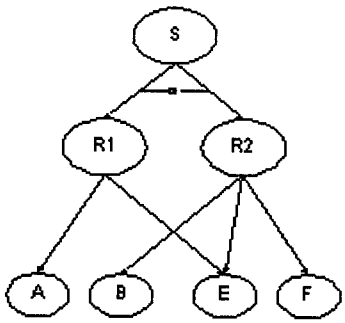
Because this specialization divides a threat into a probability that an attack will occur and a probability that a resource is affected if the attack occurs, it is possible to make use of recent history

to improve the use of the model. If some resource has been damaged by an attack and the attack is of the sort that can last through multiple time intervals (as most can), then we know that an attack is in progress and therefore P(A)=1. The result is that other resources subject to the threat are more likely to be damaged than they would be if it were not known that the threat had manifested itself as an active attack. This deals nicely with the "burning building problem". While it is generally a good idea to not place critical resources such that they can be damaged by a single fire, it is essential to avoid such configurations the fire is occurring right now. Of course, similar observations apply to other kinds of threats as well, where we might want to avoid over reliance on one platform type to avoid hacker attacks, but the need to do so becomes much more critical when such attacks are known to be occurring.

3.3. Threats to Complex Resources

A complex resource (service) is threatened by the failure of lower level resources upon which it relies. As such, it is threatened by the same threats as these lower level resources. The example below illustrates the straightforward manner of computing how threats to complex resources are determined based on threats to base resources. Optimizations are discussed in the next section.

Example #5 - Threats to Complex Resources. A service S requires at least one replica from the set {R1, R2 }. Dependencies of the replicas are:



Threats to these resources are:

$T_{destroyed}$ = ({A}, 1.0, 0.01)
T_{power} = ({A, B}, 0.01, 1.0)
T_{virus} = ({E, F}, 0.04, 0.25)

S performs if it satisfies:

S = R1 or R2 = (A and E) or (B and E and F) = E and (A or (B and F))

The only states for [ABEF] that satisfy this equation are:

[RRRR]
[RRRF]
[RFRR]
[RFRF]
[FRRR]

The damage effect matrices are shown below. Transitions not shown have P=0. N represents NotFail, F represents Fail.

$T_{destroyed}$

transition	NNNN	FNNN
probability	0.9900	0.0100

T_{power}

transition	NNNN	FFNN
probability	0.9900	0.0100

T_{virus}

transition	NNNN	NNNF	NNFN	NNFF
probability	0.9825	0.0075	0.0075	0.0025

Applying the threats to iteratively compute the probability distribution of the state space yields the following table.

state	after $T_{destroyed}$	after T_{power}	after T_{virus}
RRRR	0.990000	0.980100	0.9629483
RRRF	-	-	0.0073508
RRFR	-	-	0.0073508
RRFF	-	-	0.0024503
RFRR	-	-	-
RFRF	-	-	-
RFFR	-	-	-
RFFF	-	-	-
FRRR	0.010000	0.009900	0.0097268

FRRF	-	-	0.0000743
FRFR	-	-	0.0000743
FRFF	-	-	0.0000248
FFRR	-	0.0100000	0.0098250
FFRF	-	-	0.0000750
FFFR	-	-	0.0000750
FFFF	-	-	0.0000250

Summing the probabilities of ending in the states that satisfy S, ([RRRR], [RRRF], [RFRR], [RFRF], and [FRRR]) gives a probability of continuing to run of 0.9800258. This means that there is approximately a 2% probability of failure during the next time interval.

4. Optimizations

The calculations shown above are expensive because of the number of the number of resource group states that must be calculated. This section presents several techniques to reduce this calculation without sacrificing accuracy. At some point in the future, we hope to also consider approximate techniques that will substantially reduce the amount of computation by allowing the introduction of a bounded amount of error.

4.1. Coalescing Irrelevant States

In example #5, only 5 of the possible 16 states of the resource group allowed the service to function. Because the effect of attacks is monotonically non-increasing (i.e., things never get better as the result of a threatened attack), once a failure state has been reached, it does not matter what else happens to the system since it will still be in one of the failure states (although not necessarily the same one). We can capitalize on this observation by coalescing all failure states into a single synthetic state [OTHER]. It then is necessary to only keep track of the individual states that allow continued operation plus the one state that does not. In example #5, this is 6 states: [RRRR], [RRRF], [RFRR], [RFRF], [FRRR], and [OTHER]. The calculation for this is shown below.

Example #6: Coalescing Irrelevant States

state	after $T_{destroyed}$	after T_{power}	after T_{virus}
RRRR	0.9900	0.9801	0.96294825
RRRF	-	-	0.00735075
RFRR	-	-	-
RFRF	-	-	-
FRRR	0.0100	0.0099	0.00972675
OTHER	-	0.0100	0.01997425

The computation is equivalent to that in example #5, but much faster. The gain comes from the fact that fewer states need be considered (6 as opposed to 16). Note also that the failure probability is simply the probability of the system ending in [OTHER].

An interesting special case of this is when a service succeeds if and only if all resources in its resource group succeed. In this case, the computation is very simple. If all resources in a resource group are required, then the only state of the resource group that that allows the service to perform is [Run, Run, ... Run]. This can be reached if and only if the damage effect of every threat is [NotFail, NotFail, ... NotFail]. Since all threats are assumed to be independent, this is simply $\prod_i (P(D_i =$ [NotFail, NotFail, ..., NotFail])).

4.2. Bottom-Up Computation of Intermediate States

At any single level of services, there are typically a small number of direct resource dependencies; the troublesome state explosion occurs because of the recursive nature of the resource dependencies. Thus, if it is possible to perform a bottom-up computation of the probability of intermediate resources failing, no single computation will be too large. However, as the following example illustrates, a naive bottom-up approach can yield dramatically incorrect results.

Example #7 - Incorrect, Bottom-up Computation of Intermediate States: Consider the resource dependencies and threats from example #5. A bottom-up approach would compute the probability of failure of R1 from A and E, of R2 from B, E, and F, and then S from R1 and R2.

R1 and R2 run only if all of their resources are also running, so it is only necessary to compute that each threat causes no failures. We know that for a given threat:

$$P(\text{no failures}) = (1 - P(A)) + (P(A)*(1-P(\text{Fail}(r)|A))^n)$$

where n is the number of resources to which the threat applies

Thus:

P([A, E] = [Run, Run]) due to:

$$T_{destroyed} = (1.00 - 1.00) + (1.00*(1.00-0.01)^1) = 0.9900$$
$$T_{power} = (1.00 - 0.01) + (0.01*(1.00-1.00)^1) = 0.9900$$
$$T_{virus} = (1.00 - 0.04) + (0.04*(1.00-0.25)^1) = 0.9900$$

P([B, E, F] = [Run, Run, Run]) due to:

$$T_{destroyed} = (1.00 - 1.00) + (1.00*(1.00-0.01)^0) = 1.0000$$
$$T_{power} = (1.00 - 0.01) + (0.01*(1.00-1.00)^1) = 0.9900$$
$$T_{virus} = (1.00 - 0.04) + (0.04*(1.00-0.25)^2) = 0.9825$$

So:

P([R1] = [Run]) = 0.9900 * 0.9900 * 0.9900 = 0.970299
P([R2] = [Run]) = 1.0000 * 0.9900 * 0.9825 = 0.972675

P([R1] = [Fail]) = 1 - 0.970299 = 0.029701
P([R2] = [Fail]) = 1 - 0.972675 = 0.027325

Thus:

P([S] = [Fail]) = P([R1]=[Fail]) * P([R2]=[Fail]) = 0.029701 * 0.027325 = 0.000811579825

This method of calculation estimates approximately a 0.08% probability that S will fail in the next time interval. It is a quick computation; however, it is incorrect, as can be seen by comparing it with the correct results in example #5, which indicated a 2% probability of failure. The source of this discrepancy lies in two correlations between the resources used by R1 and R2. The most obvious correlation is that resource E is used by both replicas, so a single failure will cause both to fail. Somewhat less obvious is the fact that attacks against A and B are highly correlated (T_{power} is completely and $T_{virus}$is partially correlated) . The result is that even though these resources are not reused, they share common vulnerabilities; in this case to loss of power or viral attack.

Intermediate results can be safely computed only if all resource reuse and correlated threats are handled properly. One way to do this is to compute the state space of all base resources and from there iteratively, compute the state space of intermediate resources. This is shown below.

Example #8: Bottom-Up Computation of Intermediate States. Again, going back to example #5, we would compute the state space distributions of [ABEF], then [R1,R2], then S. These are:

State Space Distribution of [ABEF]

state	after $T_{destroyed}$	after T_{power}	after T_{virus}

RRRR	0.990000	0.980100	0.9629483
RRRF	-	-	0.0073508
RRFR	-	-	0.0073508
RRFF	-	-	0.0024503
RFRR	-	-	-
RFRF	-	-	-
RFFR	-	-	-
RFFF	-	-	-
FRRR	0.010000	0.009900	0.0097268
FRRF	-	-	0.0000743
FRFR	-	-	0.0000743
FRFF	-	-	0.0000248
FFRR	-	0.0100000	0.0098250
FFRF	-	-	0.0000750
FFFR	-	-	0.0000750
FFFF	-	-	0.0000250

State Space Distribution of [R1, R2]

state	all threats	resource states
RR	0.9629483	[RRRR]
RF	0.0073508	[RRRF], [RFRR], [RFRF]
FR	0.0097268	[FRRR]
FF	0.0199742	all others

S runs unless the state of [R1, R2] is [FF].

Correct bottom-up computation has little advantage except in two cases:

- there are many top level services whose failure probability can be computed from the same intermediate state distributions, and
- the number of base resources can be dramatically reduced by taking advantage of the shape of the resource dependency graph.

Computing for multiple onelevel services may prove useful in the future, but it imposes additional complexity on the functioning of the market. At present, our market does not do this.

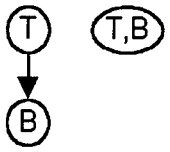
Shrinking the size of the resource group state space using the graph structure of the resource dependencies is discussed next.

4.3. Collapsing the State Space

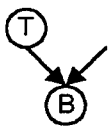
The failure probability for a high level service depends on the threats against that service and also all of its dependencies. In the absence of replica groups, the probability that a resource keeps running is the probability that it and the logical "and" of all its dependencies keep running. Replica groups act as "or" or "threshold" branches. The subgraph of dependencies for a particular resource implies a logical expression over the state space of resources. The general threat calculation computes the probability that each state will be reached. The logical expression can be evaluated over each potential state and the probabilities for all state which are false (down) are summed to yield the probability that the resource fails.

The problem with this is that the calculation of the state probabilities is exponential in the total number of resources. If the resource under consideration depends on a large number of resources the computation will quickly become intractable. Our goal is to find a less computationally intensive calculation of the failure probabilities. One possibility is to develop an approximation with an upper and lower bound. This approach has potential, but we haven't had too much success with it yet. We have had success in dramatically reducing the number of states which must be considered which yields an exact result.

Operations on the Resource Dependency Graph

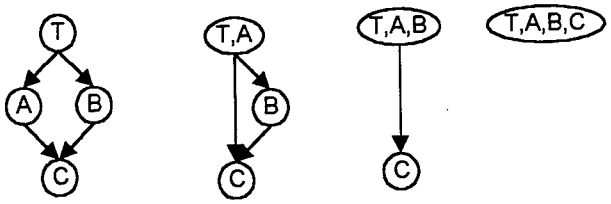


Consider the figure above. T and B are resources and T depends on B. T continues to run only if B continues to run, so the probabilities that each keep running do not need to be considered separately. For the purpose of the threat calculation we can collapse T and B into one node T,B. The threats against T and B are merged with an adjustment described later.



If there were a dependency on B from somewhere else in the subgraph (as in the figure above), the states of B and T must be considered independently. In merging T and B information about the individual states is lost. This information is needed to calculate the probability of failure for whatever is above B and its correlation to T.

If this transformation is applied in a top down manner, there will be no such outside dependency unless there are replica groups. Therefore, an entire resource dependency graph can be collapsed into a single node as shown in the diagram below.

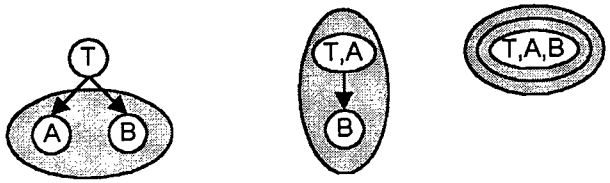


In this diagram C has two incoming dependencies an therefore cannot be merged with either A or B. However, this is not necessary. The transformation is applied in a top down, breadth first manner. Once two nodes are collapsed, the search begins again from the root node. In the first search, A is identified as a viable merger candidate and merged with T. Then the search begins again from T,A. B is identified as a merger candidate and merged to form T,A,B. At this point the dual dependencies converging on C are merged. C is now a merger candidate and is identified on the next pass. It is merged to form T,A,B,C.

Prior to the transformation, the calculation of T's failure probability would have required 4 states. Since each state can be either up or down this results $2^4 = 16$ points in the state space to consider. After the transformation there is only one state and two points in the state space.

Adjusting Threat Descriptions

The threat model models threats against multiple resources, for example power outage or a viral attack. These threats need to be adjusted if their members are merged.



Consider the diagram above. T depends on A and B. A and B are vulnerable to the same threat (the shaded area). When A is merged in to form T,A that threat then applies to B and T,A. There is no problem with this. There is a problem when B is merged to form T,A,B. The probability that the group is attacked remains the same, but the individual probability of failure can change. Consider the case that the threat group is something like vulnerability to the same virus. There is some probability that the system is exposed to the virus, say 0.2. Given that there is viral contamination, there is some probability that either A or B will fail, say 0.1. Merging A and B into T does not change the probability of viral contamination, but the probability T,A,B fails conditioned on the contamination is

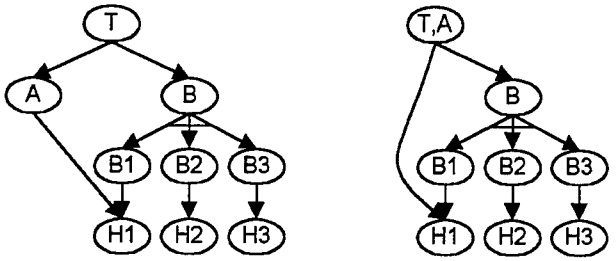
not 0.1. If the virus infects either A or B, T,A,B will fail. The virus now has two chances against the one node. The correct probability of failure given contamination is P(A or B) = P(A) + P(B) - P(A and B) = 0.1 + 0.1 - 0.01 = 0.19.

Replica Groups

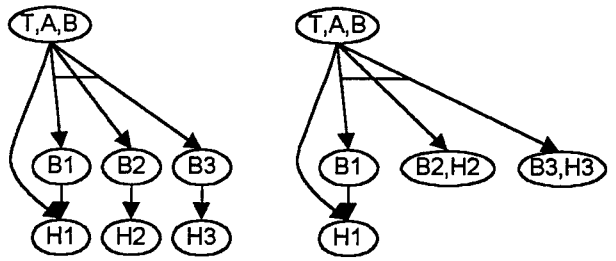
Replica groups impose complexity because their states do need to be considered independently and cannot be collapsed in general. There is some possibility that by exploiting symmetry or approximation this limitation can avoided in common special cases, but we do not have a result in this area yet. Although, we have run down some promising looking box canyons.

Putting It Together

An example demonstrating a realistic level of complexity is provided.



In the example we have a high level resource T. T depends on a service A which runs on host H1. In addition, T depends on a replica group B. B depends on three replicas B1, B2, and B3. In the diagram the bar across the links indicates an "or" dependency rather than the standard "and" dependency. B1, B2, and B3 depend on H1, H2, and H3 respectively. Applying the transformation once identifies A and merges it to form T,A.



The second application identifies B. Note that in our model we explicitly represent the threat against a coordinator of a replica group as distinct from those of the replicas. B is merged to form T,A,B which has 4 dependencies, A's dependency on H1 and B's dependencies on B1, B2, and B3.

The third and forth applications identify H2 and H3 and merge them into B2 and B3. This is the full degree of compression that we can accomplish with the simple transformation. It has compressed a graph with 9 nodes or 512 points in the state space to 5 with only 32 points in the state space.

5. Issues

5.1. Waste of Resources During Allocation

Self-adaptive systems can waste resources in at least the following ways.

- **Dedicating the wrong amount of resources to meeting application needs.** The obvious example of this would be using too many resources for some purpose. This "over engineering" assigns resources to tasks for which they are not really needed to "make sure" that the task gets done. This is not a problem if resources are plentiful, but can cause other tasks to be starved when resources dwindle. A less obvious example is if the assignment of too few resources to a task causes the task to fail to be accomplished, thereby wasting the resources that were futilely committed.

- **Excessive optimization.** Regardless of the objective function being optimized by the adaptation mechanisms, globally optimal configurations will not be achievable in practice. This means that it will usually appear that a slight "tweaking" of the system could result in a better configuration. However, this is misleading, since the model serving as the basis for optimization contains many errors and uncertainties as outlined above. This uncertainty means that once a certain level of "goodness" has been reached, it will be impossible to tell whether a given adaptation makes the situation better or worse. Bounding the uncertainty tells when further adaptation is pointless.

- **Optimizing the wrong thing.** It will be impossible to make adaptation decisions about all parts of the system simultaneously because it would be very expensive. An understanding of errors can help determine which parts of the system might be furthest from their desired state and thus are most deserving of attention.

- **Fast vs. ideal decisions.** Given the model uncertainty, is it better to make many quick, possibly quite sub-optimal adaptation decisions, or fewer, better decisions more slowly. Since decision quality will be bounded by model uncertainty, should the resources invested in reaching a decision be tunable to tailor the sophistication of the decision making to the model quality?

An understanding of errors and uncertainty (below) is necessary (but not sufficient) to avoid these kinds of mistakes.

5.2. Errors and Uncertainty

The quality of the adaptation decisions that can be made is clearly constrained by the fidelity with which the model reflects the real world. Model infidelities can lead to invalid configurations that do not function properly. They can also cause the decision process to fail to find legitimate adaptations or to perform functionally valid but sub-optimal adaptations. Modeling errors and uncertainty can arise from faulty specifications, sensing errors, errors in assumptions about future behavior, and data management inconsistencies.

To date, little work has been done to determine the effect of modeling and sensing errors on the ability of self-adaptive systems to adapt well. Additional work is needed to understand types and sources of errors and uncertainties that can occur in self-adaptive systems and understand the effect of such errors and uncertainties on the quality of the adaptation decisions that can be reached, and determine ways to ameliorate the effect of such errors and uncertainties.

Errors and uncertainty can creep into a survivable system from several sources (discussed in more detail below):

- module and physical resource specifications,
- sensed state and events,
- assumptions about future behavior,
- anomalies due to distributed models and decision making, and

5.2.1. Specification Errors

A survivable system requires specifications of a variety of things in order to be able to construct a robust configuration and correctly recover from failures. Errors in the specifications can cause undesirable behavior. Specification vulnerabilities arise from:

- **Invalid capability advertisements and requirement specifications.** Self-adaptive systems assume that the capabilities of functional modules and the resources they need to provide that functionality can be known to the processes making adaptation decisions. If these are unknown it will be impossible to physically instantiate functional units or connect them to other functionality they need. If the information is incorrect, invalid configurations are likely to result. The obvious way to obtain such information is by developer written specifications that are either stored in a repository somewhere or available by introspection from the components themselves. It is also theoretically possible to generate at least some of these specifications from the design information produced during module development, although this is definitely an open research area. It is not clear how to deal with errors of this type other than to have some sort of feedback mechanism to flag potential errors when they are discovered. For our purposes, we assume that these specifications exist and are accurate.

- **Incompatible schemata.** If the specifications described above come from many sources as will likely be the case in open systems, there is a danger that it will be impossible to accurately match requirements with capabilities. Schema incompatibility is a well known problem that we are not attempting to solve.

- **Imprecise specifications.** The amount of information in capability and requirements specifications needs to be bounded to make their production and management tractable. A danger is that if specifications are not defined in enough level of detail it will be impossible to determine whether matching can be done. An analysis of the appropriate level of detail, while outside our scope, is important.

5.2.2. Sensing Errors

The state of the adapting system is dynamic and must be monitored. Both the normal operations of the system (e.g., current loading) and exceptional events (e.g., failures) must be tracked. There are many ways in which this sensing can be done, but all are subject to error.

- **Late detection of failures.** Failures are generally not detected until after they have occurred, although in some cases symptoms of impending failure may be detected. The time lag between failure and detection may be significant depending on the type of failure and the sophistication of the detectors. Late detection may cause the system to be in a failed state for a significant time before adaptation takes place. If detection delays are significant, the adaptation mechanisms may be forced to construct conservative configurations to reduce the risk that total failure will occur faster than it can be repaired; this will consume additional resources. Estimating a bound on such delays should provide information about how much redundancy is needed to compensate.

- **Misclassified failures.** Failure detectors map observed symptoms of errors to the errors themselves. The more precise this determination can be, the better the adaptation to the failure can be. For example, if it is known that a particular host failed, a reasonable adaptation would be to migrate functionality to an identical host. However, if it were known that the host failed because of a viral attack that was specific to that class of host, such a migration would not be a good adaptation strategy. Mapping symptoms to specific failures is far from a perfect process. Assuming that failures are detected at all, they may be misclassified (e.g., determined to be viral attack when in fact it was a disk sector error) or classified too broadly (e.g., host X failed without giving a reason). It is possible, in principle to trade off between these two kinds of misclassification, accepting broader error determinations in an attempt to ensure that they are correct. It appears useful for the adaptivity decision mechanisms to know the bias of the detectors and classifiers and possibly be able to tune them.

- **Sampling errors.** Performance and loading information probably cannot be continuously sampled, since to do so would pose unacceptable overhead. Instead it is sampled; these samples are of course susceptible to sampling error. The decision process should have insight into the accuracy and timeliness of such samples. This would be useful in the following example. A reasonable strategy when adapting to meet a quality of service goal is to configure in a way that might be sub-optimal, but is less likely to have performance outside the desired range. To do this, it is necessary to bound the probability that performance will fall outside the desired range; this is function of both the variability in the performance of the resource and the uncertainty with which its performance has been measured.

5.2.3. Uncertainty in Assumptions About Future Behavior

When adaptations are performed, it is useful to consider not just the present configuration that will result, but also how that configuration is likely to behave in the future based on some expectation of future events. If a system is adapting to make a particular data source more robust by replicating it, the desirable number of replicas certainly depends on whether they are being placed on Tandem Non-Stop machines or on Windows NT machines.

These estimates of future events are "uncertain" in the sense that it is impossible to objectively determine some truth against which the estimate can be gauged. Examples of the kinds of estimation uncertainty to which this portion of the model is susceptible are:

- **Uncertainty about threat probabilities.** To avoid adapting into a configuration that is either too brittle or that wastes resources, it is useful to know what kinds of things can go wrong and how likely they are. This is termed a threat model and describes both intentional attacks and accidental failures. Some threats are reasonably well understood statistically and can probably be modeled rather accurately; failure rates for hardware are examples of this. Infrequently manifested physical threats such as fires are more problematic; their infrequency makes it hard to accurately quantify their severity. Even worse are intentional attacks whose occurrence is at the whim of an attacker who can (attempt to) mount them at times to be most disruptive; gathering statistics on these will be extremely difficult. For such attacks, it is not even clear that probabilities are the best way to describe their severity or frequency. Even if an appropriate language can be determined, it is not clear how to estimate the severity of a threat that can be imposed or withheld at will.

- **Failure to correlate failures.** Failures and attacks obviously correlate as discussed above. However, because of the difficulty of diagnosing the root causes of failures, it is possible that resource failures that result from the same vulnerability are ascribed to different causes. Conversely, it is possible that independent failures are wrongly ascribed to a common cause. At issue are how to measure or estimate such correlation and the effect of incorrect estimates.

- **Correlated attacks.** It is also possible that the occurrence of certain kinds of attacks may be an indicator that other attacks are impending and should be guarded against by an adaptation. For example, if the number of attacks against particular parts of the system increase, it might be reasonable to assume that other similar parts of the system might be attacked also, or that other kinds of attacks might be mounted against the components already under attack.

- **Time-varying threat probabilities.** The severity of threats can be situational or can change over time. Assuming a static threat model is unreasonable, but mechanisms to evolve the model need to be evaluated. A related issue is how to model and assess attacks that become more likely to succeed over time. Password guessing is a trivial example of such an attack. If it were known how to handle such kinds of attacks it might be possible to "preemptively" adapt the system before it succumbed.

5.2.4. Errors Due to Distributed Models and Decision Making

Errors or inconsistencies can introduced by the distributed nature of self-adaptive, survivable systems. The decision process and model must both be highly distributed. The distributed model cannot be kept synchronous, since such a transactional model of data management would form a choke point. The following kinds of inconsistencies may occur in a distributed environment.

- **Inconsistencies due to propagation delays.** Adaptation decisions will be made with different, possibly conflicting models due to delays in the propagation of sensed information. If model consistency is required for a particular decision, it would be useful to bound how

long information takes to propagate.

- **Inconsistencies due to variable levels of model detail.** It is unlikely that all model detail will propagate everywhere, even given arbitrary time. Localization of adaptation decisions and local control of resources (I don't get to use your computer without your permission) will tend to cause local parts of the system to be modeled at greater detail than remote parts of the system. Decisions made based on these differing views may be inconsistent. It would be useful to know in what ways and to develop strategies to have adaptation decisions made using the best available model.

- **Inconsistent adaptation decisions.** Two parts of the system may choose to independently adapt in conflicting ways. This is sometimes called "The Gift of the Magi Problem" in game theory, after the mutually detrimental behavior exhibited in the O. Henry story.

*U.S. GOVERNMENT PRINTING OFFICE: 1999-510-079-81213

DISTRIBUTION LIST

addresses	number of copies
AFRL/IFGA ATTN: PATRICK HURLEY 525 BROOKS ROAD ROME, NEW YORK 13441-4505	5
OBJECT SERVICES AND CONSULTANT, INC 6111 BAYWOOD AVENUE BALTIMORE, MD 21209-3803	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/MLME 2977 P STREET, STE 6 WRIGHT-PATTERSON AFB OH 45433-7739	1

AFRL/HESC-TDC
2698 G STREET, BLDG 190
WRIGHT-PATTERSON AFB OH 45433-7604 — 1

ATTN: SMDC IM PL
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801 — 1

TECHNICAL LIBRARY D0274(PL-TS)
SPAWARSYSCEN
53560 HULL ST.
SAN DIEGO CA 92152-5001 — 1

COMMANDER, CODE 4TL000D
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100 — 1

CDR, US ARMY AVIATION & MISSILE CMD
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000 — 2

REPORT LIBRARY
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545 — 1

ATTN: D'BORAH HART
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591 — 1

AFIWC/MSY
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016 — 1

ATTN: KAROLA M. YOURISON
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213 — 1

USAF/AIR FORCE RESEARCH LABORATORY 1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202